CampusPress Référence

Qt 4 et C++

http://www.free-livres.com/

Programmation d'interfaces GUI

Jasmin Blanchette et Mark Summerfield
Préface de Matthias Ettrich

Réseaux et télécom | **Programmation** | Génie logiciel | Sécurité | Système d'exploitation

PEARSON

Qt4 et C++

Programmation d'interfaces GUI

Jasmin Blanchette
et Mark Summerfield

CAMPUSPRESS

CampusPress a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, CampusPress n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

CampusPress ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par CampusPress
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00

Titre original : *C++ GUI programming with Qt 4*,
Traduit de l'américain par Christine Eberhardt,
Chantal Kolb, Dorothée Sittler

Mise en pages : TyPAO

ISBN original : 0-13-187249-4
Copyright © 2006 Trolltech S.A.

ISBN : 978-2-7440-4092-4
Copyright© 2009 Pearson Education France
Tous droits réservés

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2° et 3° a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

Table des matières

A propos des auteurs VII
Avant-propos IX
Préface XI
Remerciements XIII
Bref historique de Qt XV

Partie I - Qt : notions de base..... 1

CHAPITRE 1. Pour débuter 3
Hello Qt 4
Etablir des connexions 6
Disposer des widgets 7
Utiliser la documentation de référence 10

CHAPITRE 2. Créer des boîtes de dialogue 15
Dérivation de QDialog 16
Description détaillée des signaux et slots 22
Conception rapide d'une boîte de dialogue 25
Boîtes de dialogue multiformes 32
Boîtes de dialogue dynamiques 39
Classes de widgets et de boîtes de dialogue intégrées 40

CHAPITRE 3. Créer des fenêtres principales 45
Dérivation de QMainWindow 46
Créer des menus et des barres d'outils 51
Configurer la barre d'état 56
Implémenter le menu File 57
Utiliser des boîtes de dialogue 64
Stocker des paramètres 70
Documents multiples 72
Pages d'accueil 75

CHAPITRE 4. Implémenter la fonctionnalité d'application 77
Le widget central 78
Dérivation de QTableWidget 79
Chargement et sauvegarde 85
Implémenter le menu Edit 88
Implémenter les autres menus 92
Dérivation de QTableWidgetItem 96

CHAPITRE 5. Créer des widgets personnalisés 105
Personnaliser des widgets Qt 106
Dériver QWidget 108
Intégrer des widgets personnalisés avec le Qt Designer 118
Double mise en mémoire tampon 122

Partie II - Qt : niveau intermédiaire 141

CHAPITRE 6. Gestion des dispositions .. 143
Disposer des widgets sur un formulaire 144
Dispositions empilées 150
Séparateurs 152

Zones déroulantes 155
Widgets et barres d'outils ancrables 157
MDI (Multiple Document Interface) 159

CHAPITRE 7. Traitement des événements 169
Réimplémenter les gestionnaires d'événements 170
Installer des filtres d'événements 175
Rester réactif pendant un traitement intensif 178

CHAPITRE 8. Graphiques 2D et 3D 183
Dessiner avec QPainter 184
Transformations du painter 188
Affichage de haute qualité avec QImage . 197
Impression 199
Graphiques avec OpenGL 207

CHAPITRE 9. Glisser-déposer 213
Activer le glisser-déposer 214
Prendre en charge les types personnalisés de glisser 219
Gérer le presse-papiers 224

CHAPITRE 10. Classes d'af chage d'éléments 227
Utiliser les classes dédiées à l'affichage d'éléments 229
Utiliser des modèles prédéfinis 236
Implémenter des modèles personnalisés ... 241
Implémenter des délégués personnalisés .. 256

CHAPITRE 11. Classes conteneur 263
Conteneurs séquentiels 264
Conteneurs associatifs 273
Algorithmes génériques 276
Chaînes, tableaux d'octets et variants 278

CHAPITRE 12. Entrées/Sorties 287
Lire et écrire des données binaires 288
Lire et écrire du texte 294
Parcourir les répertoires 300
Intégration des ressources 302
Communication inter-processus 303

CHAPITRE 13. Les bases de données 309
Connexion et exécution de requêtes 310
Présenter les données sous une forme tabulaire 317
Implémenter des formulaires maître/détail 321

CHAPITRE 14. Gestion de réseau 329
Programmer les clients FTP 330
Programmer les clients HTTP 339
Programmer les applications client/serveur TCP 342
Envoi et réception de datagrammes UDP . 353

CHAPITRE 15. XML 359
Lire du code XML avec SAX 360
Lire du code XML avec DOM 365
Ecrire du code XML 370

CHAPITRE 16. Aide en ligne 373
Infobulles, informations d'état et aide "Qu'est-ce que c'est ?" 374
Utilisation de QTextBrowser comme moteur d'aide simple 376
Utilisation de l'assistant pour une aide en ligne puissante 379

Partie III - Qt : étude avancée 383

CHAPITRE 17. Internationalisation 385
Travailler avec Unicode 386
Créer des applications ouvertes aux traductions 390
Passer dynamiquement d'une langue à une autre 396
Traduire les applications 402

CHAPITRE 18. Environnement multithread 407
Créer des threads 408
Synchroniser des threads 411
Communiquer avec le thread principal 418
Utiliser les classes Qt dans les threads secondaires 423

CHAPITRE 19. Créer des plug-in 425
Développer Qt avec les plug-in 426
Créer des applications capables de gérer les plug-in 435
Ecrire des plug-in d'application 439

CHAPITRE 20. Fonctionnalités spéci ques à la plate-forme 443
Construire une interface avec les API natives 444
ActiveX sous Windows 448
Prendre en charge la gestion de session X11 461

CHAPITRE 21. Programmation embarquée . 469
Démarrer avec Qtopia 470

ANNEXE A. Installer Qt 477
A propos des licences 478
Installer Qt/Windows 478
Installer Qt/Mac 479
Installer Qt/X11 479

ANNEXE B. INTRODUCTION AU LANGAGE C++ POUR LES PROGRAMMEURS JAVA ET C# ..... 483.
Démarrer avec C++ 484
Principales différences de langage 489
Les types de données primitifs 489
Définitions de classe 490
Les pointeurs 497
Les références 500
Les tableaux 502
Les chaînes de caractères 505
Les énumérations 507
TypeDef 509
Conversions de type 509
Surcharge d'opérateur 512
Les types valeur 514
Variables globales et fonctions 516
Espaces de noms 518
Le préprocesseur 520
La bibliothèque C++ standard 523

Index 529

A propos des auteurs

Jasmin Blanchette

Jasmin a obtenu un diplôme d'informatique en 2001 à l'Université de Sherbrooke, Québec. Il a fait un stage chez Trolltech durant l'été 2000 en tant qu'ingénieur logiciel et travaille dans cette société depuis début 2001. En 2003, Jasmin a co-écrit *C++ GUI Programming with Qt 3*. Il assume désormais la double responsabilité de directeur de la documentation et d'ingénieur logiciel senior chez Trolltech. Il a dirigé la conception de l'outil de traduction *Qt Linguist* et il reste un acteur clé dans la conception des classes conteneur de Qt 4. Il est également le co-éditeur de *Qt Quarterly*, la lettre d'information technique de Trolltech.

Mark Summerfield

Mark a obtenu un diplôme d'informatique en 1993 à l'Université de Wales Swansea. Il s'est ensuite consacré à une année de recherche avant de se lancer dans le monde du travail. Il a travaillé pendant plusieurs années comme ingénieur logiciel pour diverses entreprises avant de rejoindre Trolltech. Il a été directeur de la documentation de Trolltech pendant près de trois ans, période pendant laquelle il a créé *Qt Quarterly* et co-écrit *C++ GUI Programming with Qt 3*. Mark détient un Qtraining.eu et est formateur et consultant indépendant spécialisé en langage C++, Qt et Python.

Avant-propos

Pourquoi Qt ? Pourquoi des programmeurs comme nous en viennent-ils à choisir Qt ? Bien entendu, les réponses sont évidentes : la compatibilité de Qt, les nombreuses fonctionnalités, les performances du C++, la disponibilité du code source, la documentation, le support technique de grande qualité et tous les autres éléments mentionnés dans les documents marketing de Trolltech. Pour finir, voici l'aspect le plus important : Qt rencontre un tel succès, parce que les programmeurs *l'apprécient*.

Pourquoi des programmeurs vont-ils apprécier une technologie et pas une autre ? Je pense personnellement que les ingénieurs logiciel aiment une technologie qui semble bonne, mais n'apprécient pas celles qui ne le sont pas. "Bonne" dans ce cas peut avoir diverses significations. Dans l'édition Qt 3 du livre, j'ai évoqué le système téléphonique de Trolltech comme étant un exemple particulièrement bon d'une technologie particulièrement mauvaise. Le système téléphonique n'était pas bon, parce qu'il nous obligeait à effectuer des tâches apparemment aléatoires qui dépendaient d'un contexte également aléatoire. Le hasard n'est pas une bonne chose. La répétition et la redondance sont d'autres aspects qui ne sont pas bons non plus. Les bons programmeurs sont fainéants. Ce que nous aimons dans l'informatique par rapport au jardinage par exemple, c'est que nous ne sommes pas contraints de faire continuellement les mêmes choses.

Laissez-moi approfondir sur ce point à l'aide d'un exemple issu du monde réel : les notes de frais. En général, ces notes de frais se présentent sous forme de tableurs complexes ; vous les remplissez et vous recevez votre argent en retour. Technologie simple me direz-vous. De plus, vu l'incitation pécuniaire, cela devrait constituer une tâche aisée pour un ingénieur expérimenté.

Toutefois, la réalité est bien différente. Alors que personne d'autre dans l'entreprise ne semble avoir de soucis pour remplir ces formulaires, les ingénieurs en ont. Et pour en avoir discuté avec des salariés d'autres entreprises, cela paraît être un comportement commun. Nous repoussons le remboursement jusqu'au dernier moment, et il arrive même parfois que nous l'oublions. Pourquoi ? Au vu de notre formulaire, c'est une procédure standard simple. La personne doit rassembler les reçus, les numéroter et insérer

ces numéros dans les champs appropriés avec la date, l'endroit, une description et le montant. La numérotation et la copie sont conçues pour faciliter la tâche, mais ce sont des opérations redondantes, sachant que la date, l'endroit, la description et le montant identifient sans aucun doute un reçu. On pourrait penser qu'il s'agit d'un bien petit travail pour obtenir un remboursement.

Le tarif journalier qui dépend du lieu de séjour est quelque peu embarrassant. Il existe des documents quelque part qui répertorient les tarifs normalisés pour les divers lieux de séjour. Vous ne pouvez pas simplement choisir "Chicago" ; vous devez rechercher le tarif correspondant à Chicago vous-même. Il en va de même pour le champ correspondant au taux de change. La personne doit trouver le taux de change en vigueur – peut-être à l'aide de Google – puis saisir le taux dans chaque champ. En clair, vous devez attendre que votre banque vous fasse parvenir une lettre dans laquelle ils vous informent du taux de change appliqué. Même si l'opération ne s'avère pas très compliquée, il n'est pas commode de rechercher toutes ces informations dans différentes sources, puis de les recopier à plusieurs endroits dans le formulaire.

La programmation peut énormément ressembler à ces notes de frais, mais en pire. Et c'est là que Qt vient à votre rescousse. Qt est différent. D'un côté, Qt est logique. D'un autre, Qt est amusant. Qt vous permet de vous concentrer sur vos tâches. Quand les architectes de Qt rencontraient un problème, ils ne cherchaient pas uniquement une solution convenable ou la solution la plus simple. Ils recherchaient la *bonne* solution, puis l'expliquaient. C'est vrai qu'ils faisaient des erreurs et que certaines de leurs décisions de conception ne résistaient pas au temps, mais ils proposaient quand même beaucoup de bonnes choses et les mauvais côtés pouvaient toujours être améliorés. Imaginez qu'un système conçu à l'origine pour mettre en relation Windows 95 et Unix/Motif unifie désormais des systèmes de bureau modernes aussi divers que Windows XP, Mac OS X et GNU/Linux, et pose les fondements de la plate-forme applicative Qtopia pour Linux Embarqué.

Bien avant que Qt ne devienne ce produit si populaire et si largement utilisé, la dévotion avec laquelle les développeurs de ce framework recherchaient les bonnes solutions rendait Qt vraiment particulier. Ce dévouement est toujours aussi fort aujourd'hui et affecte quiconque développe et assure la maintenance de Qt. Pour nous, travailler avec Qt constitue une grande responsabilité et un privilège. Nous sommes fiers de contribuer à rendre vos existences professionnelles et open source plus simples et plus agréables.

Matthias Ettrich
Oslo, Norvège
Juin 2006

Préface

Qt est un framework C++ permettant de développer des applications GUI multiplates-formes en se basant sur l'approche suivante : "Ecrire une fois, compiler n'importe où." Qt permet aux programmeurs d'employer une seule arborescence source pour des applications qui s'exécuteront sous Windows 98 à XP, Mac OS X, Linux, Solaris, HP-UX, et de nombreuses autres versions d'Unix avec X11. Les bibliothèques et outils Qt font également partie de Qtopia Core, un produit qui propose son propre système de fenêtrage au-dessus de Linux Embarqué.

Le but de ce livre est de vous apprendre à écrire des programmes GUI avec Qt 4. Le livre commence par "Hello Qt" et progresse rapidement vers des sujets plus avancés, tels que la création de widgets personnalisés et le glisser-déposer. Le code source des exemples de programmes est téléchargeable sur le site Pearson, www.pearson.fr, à la page dédiée à cet ouvrage. L'Annexe A vous explique comment installer le logiciel.

Ce manuel se divise en trois parties. La Partie I traite de toutes les notions et pratiques nécessaires pour programmer des applications GUI avec Qt. Si vous n'étudiez que cette partie, vous serez déjà en mesure d'écrire des applications GUI utiles. La Partie II aborde des thèmes importants de Qt très en détail et la Partie III propose des sujets plus spécialisés et plus avancés. Vous pouvez lire les chapitres des Parties II et III dans n'importe quel ordre, mais cela suppose que vous connaissez déjà le contenu de la Partie I.

Les lecteurs de l'édition Qt 3 de ce livre trouveront cette nouvelle édition familière tant au niveau du contenu que du style. Cette édition a été mise à jour pour profiter des nouvelles fonctionnalités de Qt 4 (y compris certaines qui sont apparues avec Qt 4.1) et pour présenter du code qui illustre les techniques de programmation Qt 4. Dans la plupart des cas, nous avons utilisé des exemples similaires à ceux de l'édition Qt 3. Les nouveaux lecteurs n'en seront pas affectés, mais ceux qui ont lu la version précédente pourront mieux se repérer dans le style plus clair, plus propre et plus expressif de Qt 4.

Cette édition comporte de nouveaux chapitres analysant l'architecture modèle/vue de Qt 4, le nouveau framework de plug-in et la programmation intégrée avec Qtopia, ainsi qu'une nouvelle annexe. Et comme dans l'édition Qt 3, nous nous sommes concentrés sur l'explication de la programmation Qt plutôt que de simplement hacher ou récapituler la documentation en ligne de Qt.

Nous avons écrit ce livre en supposant que vous connaissez les langages C++, Java ou C#. Les exemples de code utilisent un sous-ensemble du langage C++, tout en évitant de multiples fonctions C++ rarement nécessaires en programmation Qt. Là où il n'était pas possible d'éviter une construction C++ plus avancée, nous vous avons expliqué son utilisation.

Si vous connaissez déjà le langage Java ou C# mais que vous avez peu d'expérience en langage C++, nous vous recommandons de commencer par la lecture de l'Annexe B, qui vous offre une introduction au langage C++ dans le but de pouvoir utiliser ce manuel. Pour une introduction plus poussée à la programmation orientée objet en C++, nous vous conseillons les livres *C++ How to Program* de Harvey Deitel et Paul Deitel, et *C++Primer* de Stanley B. Lippman, Josée Lajoie, et Barbara E. Moo.

Qt a bâti sa réputation de framework multiplate-forme, mais en raison de son API intuitive et puissante, de nombreuses organisations utilisent Qt pour un développement sur une seule plate-forme. Adobe Photoshop Album est un exemple d'application Windows de masse écrite en Qt. De multiples logiciels sophistiqués d'entreprise, comme les outils d'animation 3D, le traitement de films numériques, la conception automatisée électronique (pour concevoir des circuits), l'exploration de pétrole et de gaz, les services financiers et l'imagerie médicale sont générés avec Qt. Si vous travaillez avec un bon produit Windows écrit en Qt, vous pouvez facilement conquérir de nouveaux marchés dans les univers Mac OS X et Linux simplement grâce à la recompilation.

Qt est disponible sous diverses licences. Si vous souhaitez concevoir des applications commerciales, vous devez acheter une licence Qt commerciale ; si vous voulez générer des programmes open source, vous avez la possibilité d'utiliser l'édition open source (GPL). Qt constitue la base sur laquelle sont conçus l'environnement KDE (K Desktop Environment) et les nombreuses applications open source qui y sont liées.

En plus des centaines de classes Qt, il existe des compagnons qui étendent la portée et la puissance de Qt. Certains de ces produits, tels que QSA (Qt Script for Applications) et les composants Qt Solutions, sont disponibles par le biais de Trolltech, alors que d'autres sont fournis par d'autres sociétés et par la communauté open source. Consultez le site **http://www.trolltech.com/products/3rdparty/** pour plus d'informations sur les compagnons Qt. Qt possède également une communauté bien établie et prospère d'utilisateurs qui emploie la liste de diffusion `qt-interest` ; voir **http://lists.trolltech.com/** pour davantage de détails.

Si vous repérez des erreurs dans ce livre, si vous avez des suggestions pour la prochaine édition ou si vous voulez simplement faire un commentaire, n'hésitez pas à nous contacter. Vous pouvez nous joindre par mail à l'adresse suivante : **qt-book@trolltech.com**. Un erratum sera disponible sur le site **http://doc.trolltech.com/qt-book-errata.html**.

Remerciements

Nous tenons tout d'abord à remercier Eirik Chambe-Eng, le président de Trolltech. Eirik ne nous a pas uniquement encouragé avec enthousiasme à écrire l'édition Qt 3 du livre, il nous a aussi permis de passer un temps considérable à l'écrire. Eirik et Haavard Nord, le chef de la direction de Trolltech, ont lu le manuscrit et l'ont approuvé. Matthias Ettrich, le développeur en chef de Trolltech, a également fait preuve d'une grande générosité. Il a allégrement accepté que nous manquions à nos tâches lorsque nous étions obnubilés par l'écriture de la première édition de ce livre et nous a fortement conseillé de sorte à adopter un bon style de programmation Qt.

Pour l'édition Qt 3, nous avons demandé à deux utilisateurs Qt, Paul Curtis et Klaus Schmidinger, de devenir nos critiques externes. Ce sont tous les deux des experts Qt très pointilleux sur les détails techniques qui ont prouvé leur sens du détail en repérant des erreurs très subtiles dans notre manuscrit et en suggérant de nombreuses améliorations. En plus de Matthias, Reginald Stadlbauer était notre plus loyal critique chez Trolltech. Ses connaissances techniques sont inestimables et il nous a appris à faire certaines choses dans Qt que nous ne pensions même pas possible.

Pour cette édition Qt 4, nous avons également profité de l'aide et du support d'Eirik, Haavard et Matthias. Klaus Schmidinger nous a aussi fait bénéficier de ses remarques et chez Trolltech, nos principaux critiques étaient Andreas Aardal Hanssen, Henrik Hartz, Vivi Glückstad Karlsen, Trenton Schulz, Andy Shaw et Pål de Vibe.

En plus des critiques mentionnés ci-dessus, nous avons reçu l'aide experte de Harald Fernengel (bases de données), Volker Hilsheimer (ActiveX), Bradley Hughes (multithread), Trond Kjernåsen (graphiques 3D et bases de données), Lars Knoll (graphiques 2D et internationalisation), Sam Magnuson (`qmake`), Marius Bugge Monsen (classes d'affichage d'éléments), Dimitri Papadopoulos (Qt/X11), Paul Olav Tvete (widgets personnalisés et programmation intégrée), Rainer Schmid (mise en réseau et XML), Amrit Pal Singh (introduction à C++) et Gunnar Sletta (graphiques 2D et traitement d'événements).

Nous tenons aussi à remercier tout particulièrement les équipes de Trolltech en charge de la documentation et du support pour avoir géré tous les problèmes liés à la documentation lorsque le livre nous prenait la majorité de notre temps, de même que les administrateurs système de Trolltech pour avoir laisser nos machines en exécution et nos réseaux en communication tout au long du projet.

Du côté de la production, Trenton Schulz a créé le CD compagnon de l'édition imprimée de cet ouvrage et Cathrine Bore de Trolltech a géré les contrats et les légalités pour notre compte. Un grand merci également à Nathan Clement pour les illustrations. Et enfin, nous remercions Lara Wysong de Pearson pour avoir aussi bien géré les détails pratiques de la production.

Bref historique de Qt

Qt a été mis à disposition du public pour la première fois en mai 1995. Il a été développé à l'origine par Haavard Nord (le chef de la direction de Trolltech) et Eirik Chambe-Eng (le président de Trolltech). Haavard et Eirik se sont rencontrés à l'Institut Norvégien de Technologie de Trondheim, d'où ils sont diplômés d'un master en informatique.

L'intérêt d'Haavard pour le développement C++ d'interfaces graphiques utilisateurs (GUI) a débuté en 1988 quand il a été chargé par une entreprise suédoise de développer un framework GUI en langage C++. Quelques années plus tard, pendant l'été 1990, Haavard et Eirik travaillaient ensemble sur une application C++ de base de données pour les images ultrasons. Le système devait pouvoir s'exécuter avec une GUI sous Unix, Macintosh et Windows. Un jour de cet été-là, Haavard et Eirik sont sortis profiter du soleil, et lorsqu'ils étaient assis sur un banc dans un parc, Haavard a dit, "Nous avons besoin d'un système d'affichage orienté objet". La discussion en résultant a posé les bases intellectuelles d'une GUI multiplate-forme orientée objet qu'ils ne tarderaient pas à concevoir.

En 1991, Haavard a commencé à écrire les classes qui deviendraient Qt, tout en collaborant avec Eirik sur la conception. L'année suivante, Eirik a eu l'idée des "signaux et slots", un paradigme simple mais puissant de programmation GUI qui est désormais adopté par de nombreux autres kits d'outils. Haavard a repris cette idée pour en produire une implémentation codée. En 1993, Haavard et Eirik ont développé le premier noyau graphique de Qt et étaient en mesure d'implémenter leurs propres widgets. A la fin de l'année, Haavard a suggéré de créer une entreprise dans le but de concevoir "le meilleur framework GUI en langage C++".

L'année 1994 a commencé sous de mauvais auspices avec deux jeunes programmeurs souhaitant s'implanter sur un marché bien établi, sans clients, avec un produit inachevé et sans argent. Heureusement, leurs épouses étaient leurs employées et pouvaient donc soutenir leurs maris pendant les deux années nécessaires selon Eirik et Haavard pour développer le produit et enfin commencer à percevoir un salaire.

La lettre "Q" a été choisie comme préfixe de classe parce que ce caractère était joli dans l'écriture Emacs de Haavard. Le "t" a été ajouté pour représenter "toolkit" inspiré par Xt, the X Toolkit. La société a été créée le 4 mars 1994, tout d'abord sous le nom Quasar Technologies, puis Troll Tech et enfin Trolltech.

En avril 1995, grâce à l'un des professeurs de Haavard, l'entreprise norvégienne Metis a signé un contrat avec eux pour développer un logiciel basé sur Qt. A cette période, Trolltech a embauché Arnt Gulbrandsen, qui, pendant six ans, a imaginé et implémenté un système de documentation ingénieux et a contribué à l'élaboration du code de Qt.

Le 20 mai 1995, Qt 0.90 a été téléchargé sur **sunsite.unc.edu**. Six jours plus tard, la publication a été annoncée sur **comp.os.linux.announce**. Ce fut la première version publique de Qt. Qt pouvait être utilisé pour un développement sous Windows et Unix, proposant la même API sur les deux plates-formes. Qt était disponible sous deux licences dès le début : une licence commerciale était nécessaire pour un développement commercial et une édition gratuite était disponible pour un développement open source. Le contrat avec Metis a permis à Trolltech de rester à flot, alors que personne n'a acheté de licence commerciale Qt pendant dix longs mois.

En mars 1996, l'Agence spatiale européenne est devenue le deuxième client Qt, en achetant dix licences commerciales. Eirik et Haavard y croyaient toujours aussi fortement et ont embauché un autre développeur. Qt 0.97 a été publié fin mai et le 24 septembre 1996, Qt 1.0 a fait son apparition. A la fin de la même année, Qt atteignait la version 1.1 : huit clients, chacun venant d'un pays différent, ont acheté 18 licences au total. Cette année a également vu la naissance du projet KDE, mené par Matthias Ettrich.

Qt 1.2 a été publié en avril 1997. Matthias Ettrich a décidé d'utiliser Qt pour concevoir un environnement KDE, ce qui a favorisé l'élévation de Qt au rang de standard *de facto* pour le développement GUI C++ sous Linux. Qt 1.3 a été publié en septembre 1997.

Matthias a rejoint Trolltech en 1998 et la dernière version principale de Qt 1, 1.40, a été conçue en septembre de la même année. Qt 2.0 a été publié en juin 1999. Qt 2 avait une nouvelle licence open source, QPL (Q Public License), conforme à la définition OSD (Open Source De?nition). En août 1999, Qt a reçu le prix LinuxWorld en tant que meilleure bibliothèque/outil. A cette période-là, Trolltech Pty Ltd (Australie) a été fondée.

Trolltech a publié Qtopia Core (appelé ensuite Qt/Embedded) en 2000. Il était conçu pour s'exécuter sur les périphériques Linux Embarqué et proposait son propre système de fenêtrage en remplacement de X11. Qt/X11 et Qtopia Core étaient désormais proposés sous la licence GNU GPL (General Public License) fortement utilisée, de même que sous des licences commerciales. A la fin de l'année 2000, Trolltech a fondé Trolltech Inc. (USA) et a publié la première version de Qtopia, une plate-forme applicative pour téléphones mobiles et PDA. Qtopia Core a remporté le prix LinuxWorld "Best Embedded Linux Solution" en 2001 et 2002, et Qtopia Phone a obtenu la même récompense en 2004.

Qt 3.0 a été publié en 2001. Qt est désormais disponible sous Windows, Mac OS X, Unix et Linux (Desktop et Embarqué). Qt 3 proposait 42 nouvelles classes et son code s'élevait à 500 000 lignes. Qt 3 a constitué une avancée incroyable par rapport à Qt 2 : un support local et

Unicode largement amélioré, un affichage de texte et une modification de widget totalement innovant et une classe d'expressions régulières de type Perl. Qt 3 a remporté le prix "Jolt Productivity Award" de Software Development Times en 2002.

En été 2005, Qt 4.0 a été publié. Avec près de 500 classes et plus de 9000 fonctions, Qt 4 est plus riche et plus important que n'importe quelle version antérieure. Il a été divisé en plusieurs bibliothèques, de sorte que les développeurs ne doivent se rattacher qu'aux parties de Qt dont ils ont besoin. Qt 4 constitue une progression significative par rapport aux versions antérieures avec diverses améliorations : un ensemble totalement nouveau de conteneurs template efficaces et faciles d'emploi, une fonctionnalité avancée modèle/vue, un framework de dessin rapide et flexible en 2D et des classes de modification et d'affichage de texte Unicode, sans mentionner les milliers de petites améliorations parmi toutes les classes Qt. Qt 4 est la première édition Qt disponible pour le développement commercial et open source sur toutes les plates-formes qu'il supporte.

Toujours en 2005, Trolltech a ouvert une agence à Beijing pour proposer aux clients de Chine et alentours un service de vente et de formation et un support technique pour Qtopia.

Depuis la création de Trolltech, la popularité de Qt n'a cessé de s'accroître et continue à gagner du terrain aujourd'hui. Ce succès est une réflexion sur la qualité de Qt et sur la joie que son utilisation procure. Pendant la dernière décennie, Qt est passé du statut de produit utilisé par quelques "connaisseurs" à un produit utilisé quotidiennement par des milliers de clients et des dizaines de milliers de développeurs open source dans le monde entier.

I

Qt : notions de base

1 *Pour débuter*
2 *Créer des boîtes de dialogue*
3 *Créer des fenêtres principales*
4 *Implémenter la fonctionnalité d'application*
5 *Créer des widgets personnalisés*

1

Pour débuter

Au sommaire de ce chapitre

- ✔ Utiliser Hello Qt
- ✔ Etablir des connexions
- ✔ Disposer des widgets
- ✔ Utiliser la documentation de référence

Ce chapitre vous présente comment combiner le langage C++ de base à la fonctionnalité fournie par Qt dans le but de créer quelques petites applications GUI (à interface utilisateur graphique). Ce chapitre introduit également deux notions primordiales concernant Qt : les "signaux et slots" et les dispositions. Dans le Chapitre 2, vous approfondirez ces points, et dans le Chapitre 3, vous commencerez à concevoir une application plus réaliste.

Si vous connaissez déjà les langages Java ou C#, mais que vous ne disposez que d'une maigre expérience concernant C++, nous vous recommandons de lire tout d'abord l'introduction à C++ en Annexe B.

Hello Qt

Commençons par un programme Qt très simple. Nous l'étudierons d'abord ligne par ligne, puis nous verrons comment le compiler et l'exécuter.

```
1  #include <QApplication>
2  #include <QLabel>

3  int main(int argc, char *argv[])
4  {
5      QApplication app(argc, argv);
6      QLabel *label = new QLabel("Hello Qt!");
7      label->show();
8      return app.exec();
9  }
```

Les lignes 1 et 2 contiennent les définitions des classes `QApplication` et `QLabel`. Pour chaque classe Qt, il y a un fichier d'en-tête qui porte le même nom (en respectant la casse) que la classe qui renferme la définition de classe.

La ligne 5 crée un objet `QApplication` pour gérer les ressources au niveau de l'application. Le constructeur `QApplication` exige `argc` et `argv` parce que Qt prend personnellement en charge quelques arguments de ligne de commande.

La ligne 6 crée un widget `QLabel` qui affiche "Hello Qt !". Dans la terminologie Qt et Unix, un *widget* est un élément visuel dans une interface utilisateur. Ce terme provient de "gadget pour fenêtres" et correspond à "contrôle" et "conteneur" dans la terminologie Windows. Les boutons, les menus et les barres de défilement sont des exemples de widgets. Les widgets peuvent contenir d'autres widgets ; par exemple, une fenêtre d'application est habituellement un widget qui comporte un `QMenuBar`, quelques `QToolBar`, un `QStatusBar` et d'autres widgets. La plupart des applications utilisent un `QMainWindow` ou un `QDialog` comme fenêtre d'application, mais Qt est si flexible que n'importe quel widget peut être une fenêtre. Dans cet exemple, le widget `QLabel` correspond à la fenêtre d'application.

La ligne 7 permet d'afficher l'étiquette. Lors de leur création, les widgets sont toujours masqués, de manière à pouvoir les personnaliser avant de les afficher et donc d'éviter le phénomène du scintillement.

La ligne 8 transmet le contrôle de l'application à Qt. A ce stade, le programme entre dans la boucle d'événement. C'est une sorte de mode d'attente où le programme attend que l'utilisateur agisse, en cliquant sur la souris ou en appuyant sur une touche par exemple. Les actions de l'utilisateur déclenchent des *événements* (aussi appelés "messages") auquel le programme peut répondre, généralement en exécutant une ou plusieurs fonctions. Par exemple, quand l'utilisateur clique sur un widget, les événements "bouton souris enfoncé" et "bouton souris relâché" sont déclenchés. A cet égard, les applications GUI diffèrent énormément des programmes par lots traditionnels, qui traitent habituellement l'entrée, produisent des résultats et s'achèvent sans intervention de quiconque.

Pour des questions de simplicité, nous n'appelons pas `delete` sur l'objet `QLabel` à la fin de la fonction `main()`. Cette petite fuite de mémoire est insignifiante dans un programme de cette taille, puisque la mémoire est récupérée de toute façon par le système d'exploitation dès qu'il se termine.

Figure 1.1
Hello sur Linux

Vous pouvez désormais tester le programme sur votre ordinateur. Vous devrez d'abord installer Qt 4.1.1 (ou une version ultérieure de Qt 4), un processus expliqué en Annexe A. A partir de maintenant, nous supposons que vous avez correctement installé une copie de Qt 4 et que le répertoire `bin` de Qt se trouve dans votre variable d'environnement `PATH`. (Sous Windows, c'est effectué automatiquement par le programme d'installation de Qt.) Vous aurez également besoin du code source du programme dans un fichier appelé `hello.cpp` situé dans un répertoire nommé `hello`. Vous pouvez saisir le code de ce programme vous-même ou le copier depuis le CD fourni avec ce livre, à partir du fichier /examples/chap01/hello/hello.cpp.

Depuis une invite de commande, placez-vous dans le répertoire `hello`, puis saisissez

```
qmake -project
```

pour créer un fichier de projet indépendant de la plate-forme (`hello.pro`), puis tapez

```
qmake hello.pro
```

pour créer un fichier Makefile du fichier de projet spécifique à la plate-forme.

Tapez `make` pour générer le programme.[1] Exécutez-le en saisissant `hello` sous Windows, `./hello` sous Unix et `open hello.app` sous Mac OS X. Pour terminer le programme, cliquez sur le bouton de fermeture dans la barre de titre de la fenêtre.

Si vous utilisez Windows et que vous avez installé Qt Open Source Edition et le compilateur MinGW, vous aurez accès à un raccourci vers la fenêtre d'invite DOS où toutes les variables d'environnement sont correctement installées pour Qt. Si vous lancez cette fenêtre, vous pouvez y compiler des applications Qt grâce à `qmake` et `make` décrits précédemment. Les exécutables produits sont placés dans les dossiers `debug` ou `release` de l'application, par exemple C:\qt-book\hello\release\hello.exe.

1. Si vous obtenez une erreur du compilateur sur `<QApplication>`, cela signifie certainement que vous utilisez une version plus ancienne de Qt. Assurez-vous d'utiliser Qt 4.1.1 ou une version ultérieure de Qt 4.

Si vous vous servez de Microsoft Visual C++, vous devrez exécuter `nmake` au lieu de `make`. Vous pouvez aussi créer un fichier de projet Visual Studio depuis `hello.pro` en tapant

```
qmake -tp vc hello.pro
```

puis concevoir le programme dans Visual Studio. Si vous travaillez avec Xcode sous Mac OS X, vous avez la possibilité de générer un projet Xcode à l'aide de la commande

```
qmake -spec macx-xcode
```

Figure 1.2
Une étiquette avec une mise en forme HTML basique

Avant de continuer avec l'exemple suivant, amusons-nous un peu : remplacez la ligne

```
QLabel *label = new QLabel("Hello Qt!");
```

par

```
QLabel *label = new QLabel("<h2><i>Hello</i> "
                           "<font color=red>Qt!</font></h2>");
```

et générez à nouveau l'application. Comme l'illustre cet exemple, il est facile d'égayer l'interface utilisateur d'une application Qt en utilisant une mise en forme simple de style HTML (voir Figure 1.2).

Etablir des connexions

Le deuxième exemple vous montre comment répondre aux actions de l'utilisateur. L'application propose un bouton sur lequel l'utilisateur peut cliquer pour quitter. Le code source ressemble beaucoup à Hello, sauf que nous utilisons un `QPushButton` en lieu et place de `QLabel` comme widget principal et que nous connectons l'action d'un utilisateur (cliquer sur un bouton) à du code.

Le code source de cette application se trouve sur le site web de Pearson, www.pearson.fr, à la page dédiée à cet ouvrage, sous /examples/chap01/quit/quit.cpp.
Voici le contenu du fichier :

```
1  #include <QApplication>
2  #include <QPushButton>

3  int main(int argc, char *argv[])
4  {
5      QApplication app(argc, argv);
6      QPushButton *button = new QPushButton("Quit");
7      QObject::connect(button, SIGNAL(clicked()),
8                       &app, SLOT(quit()));
```

```
 9      button->show();
10       return app.exec();
11   }
```

Les widgets de Qt émettent des *signaux* pour indiquer qu'une action utilisateur ou un changement d'état a eu lieu.[1] Par exemple, `QPushButton` émet un signal `clicked()` quand l'utilisateur clique sur le bouton. Un signal peut être connecté à une fonction (appelé un *slot* dans ce cas), de sorte qu'au moment où le signal est émis, le slot soit exécuté automatiquement. Dans notre exemple, nous relions le signal `clicked()` du bouton au slot `quit()` de l'objet `QApplication`. Les macros `SIGNAL()` et `SLOT()` font partie de la syntaxe ; elles sont expliquées plus en détail dans le prochain chapitre.

Figure 1.3
L'application Quit

Nous allons désormais générer l'application. Nous supposons que vous avez créé un répertoire appelé `quit` contenant `quit.cpp` (voir Figure 1.3). Exécutez `qmake` dans le répertoire `quit` pour générer le fichier de projet, puis exécutez-le à nouveau pour générer un fichier Makefile, comme suit :

```
qmake -project
qmake quit.pro
```

A présent, générez l'application et exécutez-la. Si vous cliquez sur `Quit` ou que vous appuyez sur la barre d'espace (ce qui enfonce le bouton), l'application se termine.

Disposer des widgets

Dans cette section, nous allons créer une petite application qui illustre comment utiliser les dispositions, des outils pour gérer la disposition des widgets dans une fenêtre et comment utiliser des signaux et des slots pour synchroniser deux widgets. L'application demande l'âge de l'utilisateur, qu'il peut régler par le biais d'un pointeur toupie ou d'un curseur (voir Figure 1.4).

Figure 1.4
L'application Age

L'application contient trois widgets : `QSpinBox`, `QSlider` et `QWidget`. `QWidget` correspond à la fenêtre principale de l'application. `QSpinBox` et `QSlider` sont affichés dans `QWidget` ;

1. Les signaux Qt ne sont pas liés aux signaux Unix. Dans ce livre, nous ne nous intéressons qu'aux signaux Qt.

ce sont des *enfants* de `QWidget`. Nous pouvons aussi dire que `QWidget` est le *parent* de `QSpinBox` et `QSlider`. `QWidget` n'a pas de parent puisque c'est une fenêtre de niveau le plus haut. Les constructeurs de `QWidget` et toutes ses sous-classes reçoivent un paramètre `QWidget *` qui spécifie le widget parent.

Voici le code source :

```
1  #include <QApplication>
2  #include <QHBoxLayout>
3  #include <QSlider>
4  #include <QSpinBox>

5  int main(int argc, char *argv[])
6  {
7     QApplication app(argc, argv);

8      QWidget *window = new QWidget;
9     window->setWindowTitle("Enter your age");

10     QSpinBox *spinBox = new QSpinBox;
11     QSlider *slider = new QSlider(Qt::Horizontal);
12    spinBox->setRange(0, 130);
13     slider->setRange(0, 130);

14     QObject::connect(spinBox, SIGNAL(valueChanged(int)),
15                      slider, SLOT(setValue(int)));
16     QObject::connect(slider, SIGNAL(valueChanged(int)),
17                     spinBox, SLOT(setValue(int)));
18    spinBox->setValue(35);

19     QHBoxLayout *layout = new QHBoxLayout;
20     layout->addWidget(spinBox);
21     layout->addWidget(slider);
22    window->setLayout(layout);

23     window->show();

24     return app.exec();
25 }
```

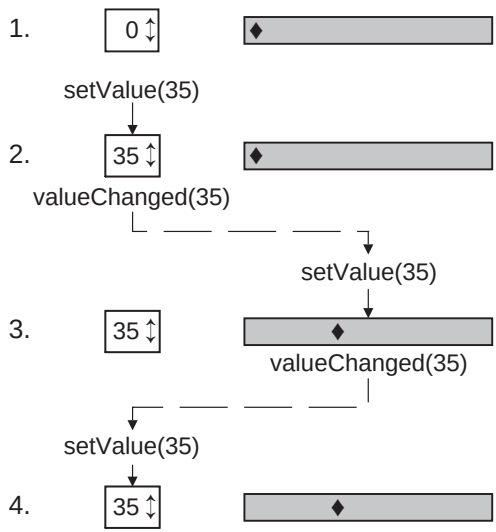
Les lignes 8 et 9 configurent `QWidget` qui fera office de fenêtre principale de l'application. Nous appelons `setWindowTitle()` pour définir le texte affiché dans la barre de titre de la fenêtre.

Les lignes 10 et 11 créent `QSpinBox` et `QSlider`, et les lignes 12 et 13 déterminent leurs plages valides. Nous pouvons affirmer que l'utilisateur est âgé au maximum de 130 ans. Nous pourrions transmettre `window` aux constructeurs de `QSpinBox` et `QSlider`, en spécifiant que le parent de ces widgets doit être `window`, mais ce n'est pas nécessaire ici, parce que le système de positionnement le déduira lui-même et définira automatiquement le parent du pointeur toupie (spin box) et du curseur, comme nous allons le voir.

Les deux appels `QObject::connect()` présentés aux lignes 14 à 17 garantissent que le pointeur toupie et le curseur sont synchronisés, de sorte qu'ils affichent toujours la même valeur. Dès que la valeur d'un widget change, son signal `valueChanged(int)` est émis et le slot `setValue(int)` de l'autre widget est appelé avec la nouvelle valeur.

La ligne 18 définit la valeur du pointeur toupie en 35. `QSpinBox` émet donc le signal `valueChanged(int)` avec un argument `int` de 35. Cet argument est transmis au slot `setValue(int)` de `QSlider`, qui définit la valeur du curseur en 35. Le curseur émet ensuite le signal `valueChanged(int)` puisque sa propre valeur a changé, déclenchant le slot `setValue(int)` du pointeur toupie. Cependant, à ce stade, `setValue(int)` n'émet aucun signal, parce que la valeur du pointeur toupie est déjà de 35. Vous évitez ainsi une récursivité infinie. La Figure 1.5 récapitule la situation.

Figure 1.5
Changer la valeur d'un widget entraîne la modification des deux widgets



Dans les lignes 19 à 22, nous positionnons le pointeur toupie et le curseur à l'aide d'un *gestionnaire de disposition (layout manager).* Ce gestionnaire est un objet qui définit la taille et la position des widgets qui se trouvent sous sa responsabilité. Qt propose trois principaux gestionnaires de disposition :

- `QHBoxLayout` dispose les widgets horizontalement de gauche à droite (de droite à gauche dans certaines cultures).
- `QVBoxLayout` dispose les widgets verticalement de haut en bas.
- `QGridLayout` dispose les widgets dans une grille.

Lorsque vous appelez `QWidget::setLayout()` à la ligne 22, le gestionnaire de disposition est installé dans la fenêtre. En arrière-plan, `QSpinBox` et `QSlider` sont "reparentés" pour devenir des enfants du widget sur lequel la disposition s'applique, et c'est pour cette raison que

nous n'avons pas besoin de spécifier un parent explicite quand nous construisons un widget qui sera inséré dans une disposition.

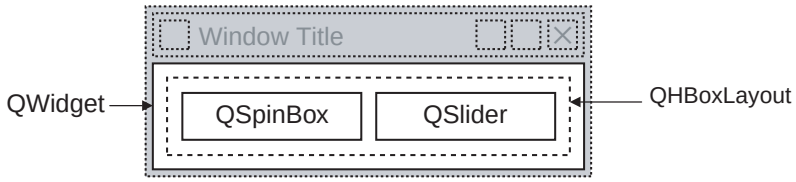


Figure 1.6
Les widgets de l'application Age

Même si nous n'avons pas défini explicitement la position ou la taille d'un widget, `QSpinBox` et `QSlider` apparaissent convenablement côte à côte. C'est parce que `QHBoxLayout` assigne automatiquement des positions et des tailles raisonnables aux widgets dont il est responsable en fonction de leurs besoins. Grâce aux gestionnaires de disposition, vous évitez la corvée de coder les positions à l'écran dans vos applications et vous êtes sûr que les fenêtres seront redimensionnées correctement.

L'approche de Qt pour ce qui concerne la conception des interfaces utilisateurs est facile à comprendre et s'avère très flexible. Habituellement, les programmeurs Qt instancient les widgets nécessaires puis définissent leurs propriétés de manière adéquate. Les programmeurs ajoutent les widgets aux dispositions, qui se chargent automatiquement de leur taille et de leur position. Le comportement de l'interface utilisateur est géré en connectant les widgets ensembles grâce aux signaux et aux slots de Qt.

Utiliser la documentation de référence

La documentation de référence de Qt est un outil indispensable pour tout développeur Qt, puisqu'elle contient toutes les classes et fonctions de cet environnement. Ce livre utilise de nombreuses classes et fonctions Qt, mais il ne les aborde pas toutes et ne fournit pas de plus amples détails sur celles qui sont évoquées. Pour profiter pleinement de Qt, vous devez vous familiariser avec la documentation de référence le plus rapidement possible.

Cette documentation est disponible en format HTML dans le répertoire doc/html de Qt et peut être affichée dans n'importe quel navigateur Web. Vous pouvez également utiliser l'*Assistant Qt*, le navigateur assistant de Qt, qui propose des fonctionnalités puissantes de recherche et d'index qui sont plus faciles et plus rapides à utiliser qu'un navigateur Web. Pour lancer l'*Assistant Qt*, cliquez sur Qt by Trolltech v4.x.y/Assistant dans le menu Démarrer sous Windows, saisissez `assistant` dans la ligne de commande sous Unix ou double-cliquez sur Assistant dans le Finder de Mac OS X (voir Figure 1.7).

Les liens dans la section "API Reference" sur la page d'accueil fournissent différents moyens de localiser les classes de Qt. La page "All Classes" répertorie chaque classe dans l'API de Qt. La page "Main Classes" regroupe uniquement les classes Qt les plus fréquemment utilisées.

En guise d'entraînement, vous rechercherez les classes et les fonctions dont nous avons parlées dans ce chapitre.

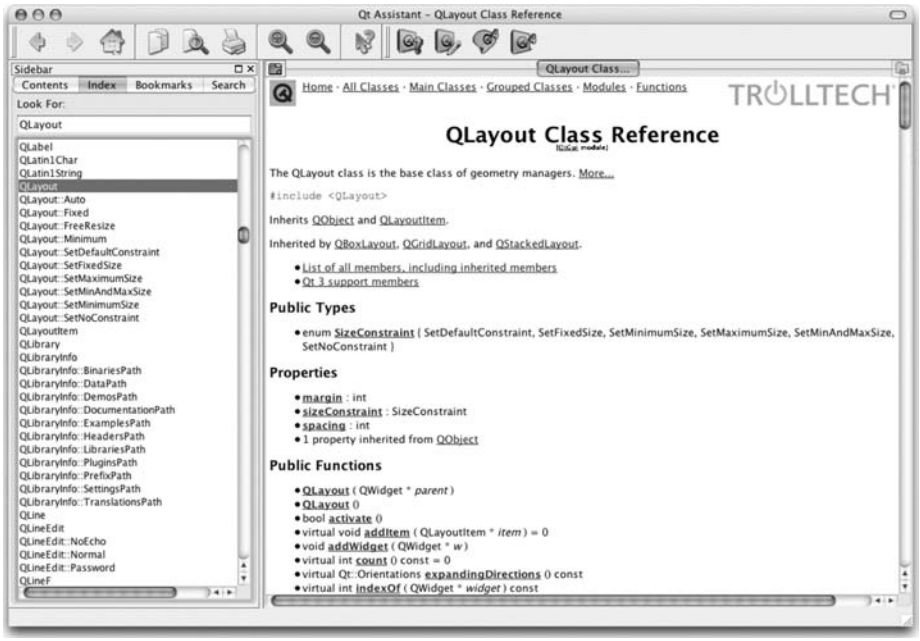
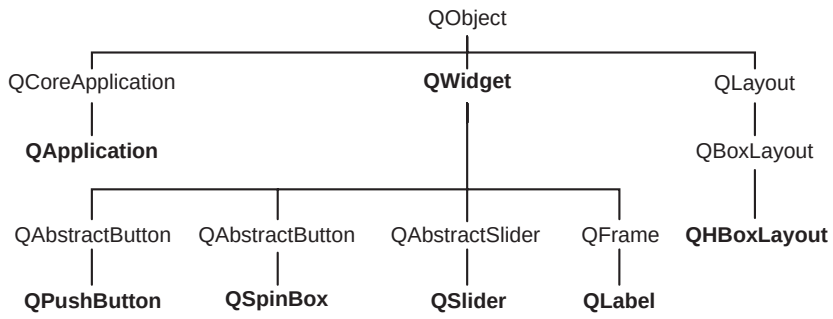


Figure 1.7
La documentation de Qt dans l'Assistant sous Mac OS X

Notez que les fonctions héritées sont détaillées dans la classe de base ; par exemple, `QPushButton` ne possède pas de fonction `show()`, mais il en hérite une de son ancêtre `QWidget`. La Figure 1.8 vous montre comment les classes que nous avons étudiées jusque là sont liées les unes aux autres.

Figure 1.8
Arbre d'héritage des classes Qt étudiées jusqu'à présent



La documentation de référence pour la version actuelle de Qt et pour certaines versions antérieures est disponible en ligne à l'adresse suivante, **http://doc.trolltech.com/**. Ce site propose également des articles sélectionnés dans *Qt Quarterly*, la lettre d'information des programmeurs Qt envoyée à tous les détenteurs de licences.

Styles des widgets

Les captures présentées jusque là ont été effectuées sous Linux, mais les applications Qt se fondent parfaitement dans chaque plate-forme prise en charge. Qt y parvient en émulant l'aspect et l'apparence de la plate-forme, au lieu d'adopter un ensemble de widgets de boîte à outils ou de plate-forme particulier.

Figure 1.9
Styles disponibles partout

Dans Qt/X11 et Qtopia Core, le style par défaut est Plastique. Il utilise des dégradés et l'anticrénelage pour proposer un aspect et une apparence modernes. Les utilisateurs de l'application Qt peuvent passer outre ce style par défaut grâce à l'option de ligne de commande `-style`. Par exemple, pour lancer l'application Age avec le style Motif sur X11, tapez simplement

```
./age -style motif
```

sur la ligne de commande.

Figure 1.10
Styles spécifiques à la plate-forme

Contrairement aux autres styles, les styles Windows XP et Mac ne sont disponibles que sur leurs plate-formes natives, puisqu'ils se basent sur les générateurs de thème de ces plate-formes.

Ce chapitre vous a présenté les concepts essentiels des connexions signal – slot et des dispositions. Il a également commencé à dévoiler l'approche totalement orientée objet et cohérente de Qt concernant la construction et l'utilisation des widgets. Si vous parcourez la documentation de Qt, vous découvrirez que l'approche s'avère homogène. Vous comprendrez donc beaucoup plus facilement comment utiliser de nouveaux widgets et vous verrez aussi que les noms choisis minutieusement pour les fonctions, les paramètres, les énumérations, etc. rendent la programmation dans Qt étonnamment plaisante et aisée.

Les chapitres suivants de la Partie I se basent sur les notions fondamentales étudiées ici et vous expliquent comment créer des applications GUI complètes avec des menus, des barres d'outils, des fenêtres de document, une barre d'état et des boîtes de dialogue, en plus des fonctionnalités sous-jacentes permettant de lire, traiter et écrire des fichiers.

2

Créer des boîtes de dialogue

Au sommaire de ce chapitre

- ✔ Dérivation de QDialog
- ✔ Description détaillée des signaux et slots
- ✔ Conception rapide d'une boîte de dialogue
- ✔ Boîtes de dialogue multiformes
- ✔ Boîtes de dialogue dynamiques
- ✔ Classes de widgets et de boîtes de dialogue intégrées

Dans ce chapitre, vous allez apprendre à créer des boîtes de dialogue à l'aide de Qt. Celles-ci présentent diverses options et possibilités aux utilisateurs et leur permettent de définir les valeurs des options et de faire des choix. On les appelle des boîtes de dialogue, puisqu'elles donnent la possibilité aux utilisateurs et aux applications de "discuter".

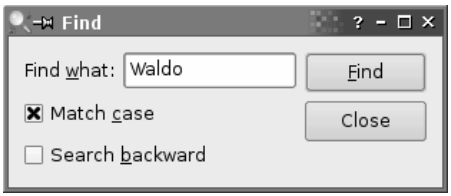
La majorité des applications GUI consiste en une fenêtre principale équipée d'une barre de menus et d'outils, à laquelle on ajoute des douzaines de boîtes de dialogue. Il est également possible de créer des applications "boîte de dialogue" qui répondent directement aux choix de l'utilisateur en accomplissant les actions appropriées (par exemple, une application de calculatrice).

Nous allons créer notre première boîte de dialogue en écrivant complètement le code pour vous en expliquer le fonctionnement. Puis nous verrons comment concevoir des boîtes de dialogue grâce au *Qt Designer*, un outil de conception de Qt. Avec le *Qt Designer*, vous codez beaucoup plus rapidement et il est plus facile de tester les différentes conceptions et de les modifier par la suite.

Dérivation de QDialog

Notre premier exemple est une boîte de dialogue Find écrite totalement en langage C++ (voir Figure 2.1). Nous l'implémenterons comme une classe à part entière. Ainsi, elle deviendra un composant indépendant et autonome, comportant ses propres signaux et slots.

Figure 2.1
La boîte de dialogue Find



Le code source est réparti entre deux fichiers : `finddialog.h` et `finddialog.cpp`. Nous commencerons par `finddialog.h`.

```
1   #ifndef FINDDIALOG_H
2   #define FINDDIALOG_H

3   #include <QDialog>

4   class QCheckBox;
5   class QLabel;
6   class QLineEdit;
7   class QPushButton;
```

Les lignes 1 et 2 (et 27) protègent le fichier d'en-tête contre les inclusions multiples.

La ligne 3 contient la définition de `QDialog`, la classe de base pour les boîtes de dialogue dans Qt. `QDialog` hérite de `QWidget`.

Les lignes 4 à 7 sont des déclarations préalables des classes Qt que nous utiliserons pour implémenter la boîte de dialogue. Une *déclaration préalable* informe le compilateur C++ qu'une classe existe, sans donner tous les détails d'une définition de classe (généralement située dans un fichier d'en-tête). Nous en parlerons davantage dans un instant.

Nous définissons ensuite `FindDialog` comme une sous-classe de `QDialog` :

```
8   class FindDialog : public QDialog
9   {
10      Q_OBJECT
```

```
11  public:
12      FindDialog(QWidget *parent = 0);
```

La macro `Q_OBJECT` au début de la définition de classe est nécessaire pour toutes les classes qui définissent des signaux ou des slots.

Le constructeur de `FindDialog` est typique des classes Qt de widgets. Le paramètre `parent` spécifie le widget parent. Par défaut, c'est un pointeur nul, ce qui signifie que la boîte de dialogue n'a pas de parent.

```
13  signals:
14      void findNext(const QString &str, Qt::CaseSensitivity cs);
15      void findPrevious(const QString &str, Qt::CaseSensitivity cs);
```

La section des signaux déclare deux signaux que la boîte de dialogue émet quand l'utilisateur clique sur le bouton Find. Si l'option Search backward est activée, la boîte de dialogue émet `findPrevious()` ; sinon elle émet `findNext()`.

Le mot-clé `signals` est en fait une macro. Le préprocesseur C++ la convertit en langage C++ standard avant que le compilateur ne la voie. `Qt::CaseSensitivity` est un type énumération qui peut prendre les valeurs `Qt::CaseSensitive` et `Qt::CaseInsensitive`.

```
16  private slots:
17      void findClicked();
18      void enableFindButton(const QString &text);

19  private:
20      QLabel *label;
21      QLineEdit *lineEdit;
22      QCheckBox *caseCheckBox;
23      QCheckBox *backwardCheckBox;
24      QPushButton *findButton;
25      QPushButton *closeButton;
26  };

27  #endif
```

Dans la section privée de la classe, nous déclarons deux slots. Pour implémenter les slots, vous devez avoir accès à la plupart des widgets enfants de la boîte de dialogue, nous conservons donc également des pointeurs vers eux. Le mot-clé `slots`, comme `signals`, est une macro qui se développe pour produire du code que le compilateur C++ peut digérer.

S'agissant des variables privées, nous avons utilisé les déclarations préalables de leurs classes. C'était possible puisque ce sont toutes des pointeurs et que nous n'y accédons pas dans le fichier d'en-tête, le compilateur n'a donc pas besoin des définitions de classe complètes. Nous aurions pu inclure les fichiers d'en-tête importants (`<QCheckBox>`, `<QLabel>`, etc.), mais la compilation se révèle plus rapide si vous utilisez les déclarations préalables dès que possible.

Nous allons désormais nous pencher sur `finddialog.cpp` qui contient l'implémentation de la classe `FindDialog`.

```
1  #include <QtGui>

2  #include "finddialog.h"
```

Nous incluons d'abord `<QtGui>`, un fichier d'en-tête qui contient la définition des classes GUI de Qt. Qt est constitué de plusieurs modules, chacun d'eux se trouvant dans sa propre bibliothèque. Les modules les plus importants sont *QtCore*, *QtGui*, *QtNetwork*, *QtOpenGL*, *QtSql*, *QtSvg* et *QtXml*. Le fichier d'en-tête `<QtGui>` renferme la définition de toutes les classes qui font partie des mo dules *QtCore* et *QtGui*. En incluant cet en-tête, vous évitez la tâche fastidieuse d'inclure chaque classe séparément.

Dans `filedialog.h`, au lieu d'inclure `<QDialog>` et d'utiliser des déclarations préalables pour `QCheckBox`, `QLabel`, `QLineEdit` et `QPushButton`, nous aurions simplement pu spécifier `<QtGui>`. Toutefois, il est généralement malvenu d'inclure un fichier d'en-tête si grand depuis un autre fichier d'en-tête, notamment dans des applications plus importantes.

```
3  FindDialog::FindDialog(QWidget *parent)
4      : QDialog(parent)
5  {
6      label = new QLabel(tr("Find &what:"));
7      lineEdit = new QLineEdit;
8      label->setBuddy(lineEdit);

9      caseCheckBox = new QCheckBox(tr("Match &case"));
10     backwardCheckBox = new QCheckBox(tr("Search &backward"));

11     findButton = new QPushButton(tr("&Find"));
12     findButton->setDefault(true);
13     findButton->setEnabled(false);

14     closeButton = new QPushButton(tr("Close"));
```

A la ligne 4, nous transmettons le paramètre `parent` au constructeur de la classe de base. Puis nous créons les widgets enfants. Les appels de la fonction `tr()` autour des littéraux chaîne les marquent dans l'optique d'une traduction en d'autres langues. La fonction est déclarée dans `QObject` et chaque sous-classe qui contient la macro `Q_OBJECT`. Il est recommandé de prendre l'habitude d'encadrer les chaînes visibles par l'utilisateur avec `tr()`, même si vous n'avez pas l'intention de faire traduire vos applications en d'autres langues dans l'immédiat. La traduction des applications Qt est abordée au Chapitre 17.

Dans les littéraux chaîne, nous utilisons le caractère & pour indiquer des raccourcis clavier. Par exemple, la ligne 11 crée un bouton Find que l'utilisateur peut activer en appuyant sur Alt+F sur les plates-formes qui prennent en charge les raccourcis clavier. Le caractère & peut également être employé pour contrôler le focus, c'est-à-dire l'élément actif : à la ligne 6, nous créons une étiquette avec un raccourci clavier (Alt+W), et à la ligne 8, nous définissons l'éditeur de lignes comme widget compagnon de l'étiquette. Ce *compagnon (buddy)* est un widget

qui reçoit le focus quand vous appuyez sur le raccourci clavier de l'étiquette. Donc, quand l'utilisateur appuie sur Alt+W (le raccourci de l'étiquette), l'éditeur de lignes reçoit le contrôle.

A la ligne 12, nous faisons du bouton Find le bouton par défaut de la boîte de dialogue en appelant `setDefault(true)`. Le bouton par défaut est le bouton qui est pressé quand l'utilisateur appuie sur Entrée. A la ligne 13, nous désactivons le bouton Find. Quand un widget est désactivé, il apparaît généralement grisé et ne répondra pas en cas d'interaction de l'utilisateur.

```
15    connect(lineEdit, SIGNAL(textChanged(const QString &)),
16            this, SLOT(enableFindButton(const QString &)));
17    connect(findButton, SIGNAL(clicked()),
18            this, SLOT(findClicked()));
19    connect(closeButton, SIGNAL(clicked()),
20            this, SLOT(close()));
```

Le slot privé `enableFindButton(const QString &)` est appelé dès que le texte change dans l'éditeur de lignes. Le slot privé `findClicked()` est invoqué lorsque l'utilisateur clique sur le bouton Find. La boîte de dialogue se ferme si l'utilisateur clique sur Close. Le slot `close()` est hérité de `QWidget` et son comportement par défaut consiste à masquer le widget (sans le supprimer). Nous allons étudier le code des slots `enableFindButton()` et `findClicked()` ultérieurement.

Etant donné que `QObject` est l'un des ancêtres de `FindDialog`, nous pouvons omettre le préfixe `QObject::` avant les appels de `connect()`.

```
21    QHBoxLayout *topLeftLayout = new QHBoxLayout;
22    topLeftLayout->addWidget(label);
23    topLeftLayout->addWidget(lineEdit);

24    QVBoxLayout *leftLayout = new QVBoxLayout;
25    leftLayout->addLayout(topLeftLayout);
26    leftLayout->addWidget(caseCheckBox);
27    leftLayout->addWidget(backwardCheckBox);

28    QVBoxLayout *rightLayout = new QVBoxLayout;
29    rightLayout->addWidget(findButton);
30    rightLayout->addWidget(closeButton);
31    rightLayout->addStretch();

32    QHBoxLayout *mainLayout = new QHBoxLayout;
33    mainLayout->addLayout(leftLayout);
34    mainLayout->addLayout(rightLayout);
35    setLayout(mainLayout);
```

Nous disposons ensuite les widgets enfants à l'aide des gestionnaires de disposition. Les dispositions peuvent contenir des widgets et d'autres dispositions. En imbriquant `QHBoxLayout`, `QVBoxLayout` et `QGridLayout` dans diverses combinaisons, il est possible de concevoir des boîtes de dialogue très sophistiquées.

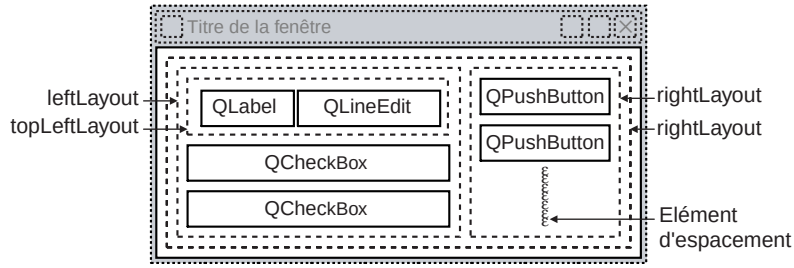


Figure 2.2
Les dispositions de la boîte de dialogue Find

Pour la boîte de dialogue Find, nous employons deux `QHBoxLayout` et deux `QVBoxLayout`, comme illustré en Figure 2.2. La disposition externe correspond à la disposition principale ; elle est installée sur `FindDialog` à la ligne 35 et est responsable de toute la zone de la boîte de dialogue. Les trois autres dispositions sont des sous-dispositions. Le petit "ressort" en bas à droite de la Figure 2.2 est un élément d'espacement (ou "étirement"). Il comble l'espace vide sous les boutons Find et Close, ces boutons sont donc sûrs de se trouver en haut de leur disposition.

Les gestionnaires de disposition présentent une subtilité : ce ne sont pas des widgets. Ils héritent de `QLayout`, qui hérite à son tour de `QObject`. Dans la figure, les widgets sont représentés par des cadres aux traits pleins et les dispositions sont illustrées par des cadres en pointillés pour mettre en avant la différence qui existe entre eux. Dans une application en exécution, les dispositions sont invisibles.

Quand les sous-dispositions sont ajoutées à la disposition parent (lignes 25, 33 et 34), les sous-dispositions sont automatiquement reparentées. Puis, lorsque la disposition principale est installée dans la boîte de dialogue (ligne 35), elle devient un enfant de cette dernière et tous les widgets dans les dispositions sont reparentés pour devenir des enfants de la boîte de dialogue. La hiérarchie parent-enfant ainsi obtenue est présentée en Figure 2.3.

```
36      setWindowTitle(tr("Find"));
37      setFixedHeight(sizeHint().height());
38  }
```

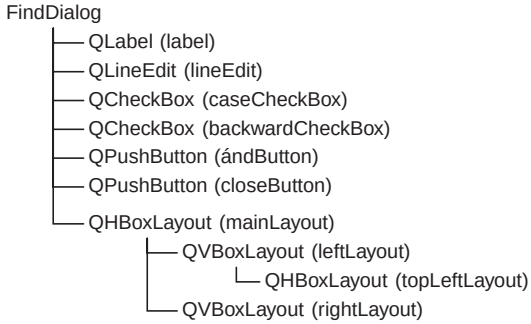


Figure 2.3
Les relations parent-enfant de la boîte de dialogue Find

Enfin, nous définissons le titre à afficher dans la barre de titre de la boîte de dialogue et nous configurons la fenêtre pour qu'elle présente une hauteur fixe, étant donné qu'elle ne contient

aucun widget qui peut occuper de l'espace supplémentaire verticalement. La fonction `QWidget::sizeHint()` retourne la taille "idéale" d'un widget.

Ceci termine l'analyse du constructeur de `FindDialog`. Vu que nous avons créé les widgets et les dispositions de la boîte de dialogue avec `new`, il semblerait logique d'écrire un destructeur qui appelle `delete` sur chaque widget et disposition que nous avons créés. Toutefois, ce n'est pas nécessaire, puisque Qt supprime automatiquement les objets enfants quand le parent est détruit, et les dispositions et widgets enfants sont tous des descendants de `FindDialog`.

Nous allons à présent analyser les slots de la boîte de dialogue :

```
39  void FindDialog::findClicked()
40  {
41     QString text = lineEdit->text();
42     Qt::CaseSensitivity cs =
43           caseCheckBox->isChecked() ? Qt::CaseSensitive
44                                     : Qt::CaseInsensitive;
45     if (backwardCheckBox->isChecked()) {
46        emit findPrevious(text, cs);
47     } else {
48        emit findNext(text, cs);
49     }
50  }

51  void FindDialog::enableFindButton(const QString &text)
52  {
53     findButton->setEnabled(!text.isEmpty());
54  }
```

Le slot `findClicked()` est appelé lorsque l'utilisateur clique sur le bouton Find. Il émet le signal `findPrevious()` ou `findNext()`, en fonction de l'option Search backward. Le mot-clé `emit` est spécifique à Qt ; comme les autres extensions Qt, il est converti en langage C++ standard par le préprocesseur C++.

Le slot `enableFindButton()` est invoqué dès que l'utilisateur modifie le texte dans l'éditeur de lignes. Il active le bouton s'il y a du texte dans cet éditeur, sinon il le désactive.

Ces deux slots complètent la boîte de dialogue. Nous avons désormais la possibilité de créer un fichier `main.cpp` pour tester notre widget `FindDialog` :

```
1   #include <QApplication>

2   #include "finddialog.h"

3   int main(int argc, char *argv[])
4   {
5       QApplication app(argc, argv);
6       FindDialog *dialog = new FindDialog;
7       dialog->show();
8       return app.exec();
9   }
```

Pour compiler le programme, exécutez `qmake` comme d'habitude. Vu que la définition de la classe `FindDialog` comporte la macro `Q_OBJECT`, le fichier Makefile généré par `qmake`

contiendra des règles particulières pour exécuter `moc`, le compilateur de méta-objets de Qt. (Le système de méta-objets de Qt est abordé dans la prochaine section.)

Pour que `moc` fonctionne correctement, vous devez placer la définition de classe dans un fichier d'en-tête, séparé du fichier d'implémentation. Le code généré par `moc` inclut ce fichier d'en-tête et ajoute une certaine "magie" C++.

`moc` doit être exécuté sur les classes qui se servent de la macro `Q_OBJECT.` Ce n'est pas un problème parce que `qmake` ajoute automatiquement les règles nécessaires dans le fichier Makefile. Toutefois, si vous oubliez de générer à nouveau votre fichier Makefile à l'aide de `qmake` et que `moc` n'est pas exécuté, l'éditeur de liens indiquera que certaines fonctions sont déclarées mais pas implémentées. Les messages peuvent être plutôt obscurs. GCC produit des avertissements comme celui-ci :

```
finddialog.o: In function 'FindDialog::tr(char const*, char const*)':
/usr/lib/qt/src/corelib/global/qglobal.h:1430: undefined reference to
'FindDialog::staticMetaObject'
```

La sortie de Visual C++ commence ainsi :

```
finddialog.obj : error LNK2001: unresolved external symbol
"public:~virtual int __thiscall MyClass::qt_metacall(enum QMetaObject
::Call,int,void * *)"
```

Si vous vous trouvez dans ce cas, exécutez à nouveau `qmake` pour mettre à jour le fichier Makefile, puis générez à nouveau l'application.

Exécutez maintenant le programme. Si des raccourcis clavier apparaissent sur votre plate-forme, vérifiez que les raccourcis Alt+W, Alt+C, Alt+B et Alt+F déclenchent le bon comportement. Appuyez sur la touche de tabulation pour parcourir les widgets en utilisant le clavier. L'ordre de tabulation par défaut est l'ordre dans lequel les widgets ont été créés. Vous pouvez le modifier grâce à `QWidget::setTabOrder()`.

Proposer un ordre de tabulation et des raccourcis clavier cohérents permet aux utilisateurs qui ne veulent pas (ou ne peuvent pas) utiliser une souris de profiter pleinement de l'application. Les dactylos apprécieront également de pouvoir tout contrôler depuis le clavier.

Dans le Chapitre 3, nous utiliserons la boîte de dialogue Find dans une application réelle, et nous connecterons les signaux `findPrevious()` et `findNext()` à certains slots.

Description détaillée des signaux et slots

Le mécanisme des signaux et des slots est une notion fondamentale en programmation Qt. Il permet au programmeur de l'application de relier des objets sans que ces objets ne sachent quoi que ce soit les uns sur les autres. Nous avons déjà connecté certains signaux et slots ensemble, déclaré nos propres signaux et slots, implémenté nos slots et émis nos signaux. Etudions désormais ce mécanisme plus en détail.

Les slots sont presque identiques aux fonctions membres ordinaires de C++. Ils peuvent être virtuels, surchargés, publics, protégés ou privés, être invoqués directement comme toute autre fonction membre C++, et leurs paramètres peuvent être de n'importe quel type. La différence est qu'un slot peut aussi être connecté à un signal, auquel cas il est automatiquement appelé à chaque fois que le signal est émis.

Voici la syntaxe de l'instruction `connect()` :

```
connect(sender, SIGNAL(signal), receiver, SLOT(slot));
```

où `sender` et `receiver` sont des pointeurs vers `QObject` et où `signal` et `slot` sont des signatures de fonction sans les noms de paramètre. Les macros `SIGNAL()` et `SLOT()` convertissent leur argument en chaîne.

Dans les exemples étudiés jusque là, nous avons toujours connecté les signaux aux divers slots. Il existe d'autres possibilités à envisager.

- **Un signal peut être connecté à plusieurs slots :**

```
connect(slider, SIGNAL(valueChanged(int)),
        spinBox, SLOT(setValue(int)));
connect(slider, SIGNAL(valueChanged(int)),
        this, SLOT(updateStatusBarIndicator(int)));
```

  Quand le signal est émis, les slots sont appelés les uns après les autres, dans un ordre non spécifié.

- **Plusieurs signaux peuvent être connectés au même slot :**

```
connect(lcd, SIGNAL(overflow()),
        this, SLOT(handleMathError()));
connect(calculator, SIGNAL(divisionByZero()),
        this, SLOT(handleMathError()));
```

  Quand l'un des signaux est émis, le slot est appelé.

- **Un signal peut être connecté à un autre signal :**

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SIGNAL(updateRecord(const QString &)));
```

  Quand le premier signal est émis, le second est également émis. En dehors de cette caractéristique, les connexions signal-signal sont en tout point identiques aux connexions signal-slot.

- **Les connexions peuvent être supprimées :**

```
disconnect(lcd, SIGNAL(overflow()),
           this, SLOT(handleMathError()));
```

  Vous n'en aurez que très rarement besoin, parce que Qt supprime automatiquement toutes les connexions concernant un objet quand celui-ci est supprimé.

Pour bien connecter un signal à un slot (ou à un autre signal), ils doivent avoir les mêmes types de paramètre dans le même ordre :

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
        this, SLOT(processReply(int, const QString &)));
```

Exceptionnellement, si un signal comporte plus de paramètres que le slot auquel il est connecté, les paramètres supplémentaires sont simplement ignorés :

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
        this, SLOT(checkErrorCode(int)));
```

Si les types de paramètre sont incompatibles, ou si le signal ou le slot n'existe pas, Qt émettra un avertissement au moment de l'exécution si l'application est générée en mode débogage. De même, Qt enverra un avertissement si les noms de paramètre se trouvent dans les signatures du signal ou du slot.

Jusqu'à présent, nous n'avons utilisé que des signaux et des slots avec des widgets. Cependant, le mécanisme en soi est implémenté dans `QObject` et ne se limite pas à la programmation d'interfaces graphiques utilisateurs. Il peut être employé par n'importe quelle sous-classe de `QObject` :

```
class Employee : public QObject
{
   Q_OBJECT

public:
   Employee() { mySalary = 0; }

   int salary() const { return mySalary; }

public slots:
   void setSalary(int newSalary);

signals:
   void salaryChanged(int newSalary);

private:
   int mySalary;
};

void Employee::setSalary(int newSalary)
{
   if (newSalary != mySalary) {
      mySalary = newSalary;
      emit salaryChanged(mySalary);
   }
}
```

Vous remarquerez la manière dont le slot `setSalary()` est implémenté. Nous n'émettons le signal `salaryChanged()` que si `newSalary != mySalary`. Vous êtes donc sûr que les connexions cycliques ne débouchent pas sur des boucles infinies.

Système de méta-objets de Qt

L'une des améliorations majeures de Qt a été l'introduction dans le langage C++ d'un mécanisme permettant de créer des composants logiciels indépendants qui peuvent être reliés les uns aux autres sans qu'ils ne sachent absolument rien sur les autres composants auxquels ils sont connectés.

Ce mécanisme est appelé *système de méta-objets* et propose deux services essentiels : les signaux-slots et l'introspection. La fonction d'introspection est nécessaire pour implémenter des signaux et des slots et permet aux programmeurs d'applications d'obtenir des "méta-informations" sur les sous-classes de `QObject` à l'exécution, y compris la liste des signaux et des slots pris en charge par l'objet et le nom de sa classe. Le mécanisme supporte également des propriétés (pour le *Qt Designer*) et des traductions de texte (pour l'internationalisation), et il pose les fondements de QSA (Qt Script for Applications).

Le langage C++ standard ne propose pas de prise en charge des méta-informations dynamiques nécessaires pour le système de méta-objets de Qt. Qt résout ce problème en fournissant un outil, `moc`, qui analyse les définitions de classe `Q_OBJECT` et rend les informations disponibles par le biais de fonctions C++. Etant donné que `moc` implémente toute sa fonctionnalité en utilisant un langage C++ pur, le système de méta-objets de Qt fonctionne avec n'importe quel compilateur C++.

Ce mécanisme fonctionne de la manière suivante :

- La macro `Q_OBJECT` déclare certaines fonctions d'introspection qui doivent être implémentées dans chaque sous-classe de `QObject` : `metaObject()`, `tr()`, `qt_metacall()` et quelques autres.
- L'outil `moc` de Qt génère des implémentations pour les fonctions déclarées par `Q_OBJECT` et pour tous les signaux.
- Les fonctions membres de `QObject`, telles que `connect()` et `disconnect()`, utilisent les fonctions d'introspection pour effectuer leurs tâches.

Tout est géré automatiquement par `qmake`, `moc` et `QObject`, vous ne vous en souciez donc que très rarement. Néanmoins, par curiosité, vous pouvez parcourir la documentation de la classe `QMetaObject` et analyser les fichiers sources C++ générés par `moc` pour découvrir comment fonctionne l'implémentation.

Conception rapide d'une boîte de dialogue

Qt est conçu pour être agréable et intuitif à écrire, et il n'est pas inhabituel que des programmeurs développent des applications Qt complètes en saisissant la totalité du code source C++. De nombreux programmeurs préfèrent cependant utiliser une approche visuelle pour concevoir

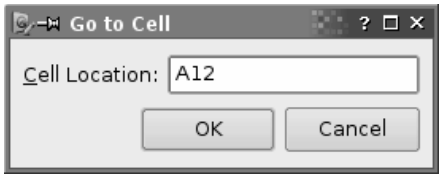
les formulaires. Ils la trouvent en effet plus naturelle et plus rapide, et ils veulent être en mesure de tester et de modifier leurs conceptions plus rapidement et facilement qu'avec des formulaires codés manuellement.

Le *Qt Designer* développe les options à disposition des programmeurs en proposant une fonctionnalité visuelle de conception. Le *Qt Designer* peut être employé pour développer tous les formulaires d'une application ou juste quelques-uns. Les formulaires créés à l'aide du *Qt Designer* étant uniquement constitués de code C++, le *Qt Designer* peut être utilisé avec une chaîne d'outils traditionnelle et n'impose aucune exigence particulière au compilateur.

Dans cette section, nous utiliserons le *Qt Designer* afin de créer la boîte de dialogue Go-to-Cell présentée en Figure 2.4. Quelle que soit la méthode de conception choisie, la création d'une boîte de dialogue implique toujours les mêmes étapes clés :

- créer et initialiser les widgets enfants ;
- placer les widgets enfants dans des dispositions ;
- définir l'ordre de tabulation ;
- établir les connexions signal-slot ;
- implémenter les slots personnalisés de la boîte de dialogue.

Figure 2.4
La boîte de dialogue Go-to-Cell

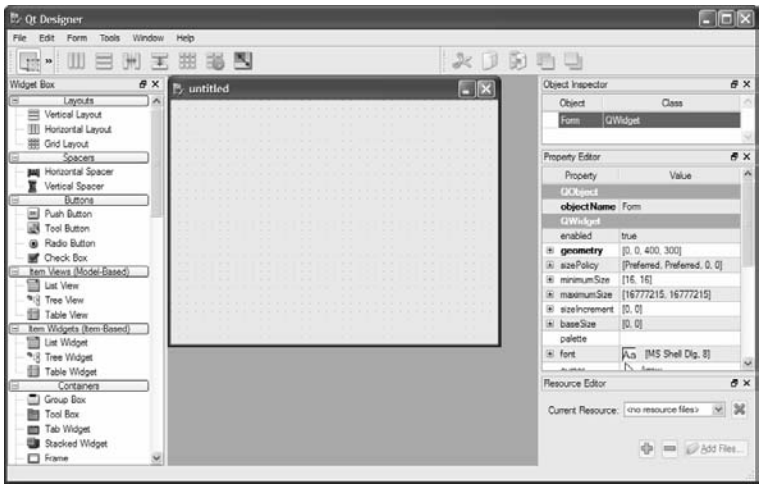


Pour lancer le *Qt Designer*, cliquez sur Qt by Trolltech v4.x.y > Designer dans le menu Démarrer sous Windows, saisissez `designer` dans la ligne de commande sous Unix ou double-cliquez sur Designer dans le Finder de Mac OS X. Quand le *Qt Designer* démarre, une liste de modèles s'affiche. Cliquez sur le modèle "Widget", puis sur OK. (Le modèle "Dialog with Buttons Bottom" peut être tentant, mais pour cet exemple nous créerons les boutons OK et Cancel manuellement pour vous expliquer le processus.) Vous devriez à présent vous trouver dans une fenêtre appelée "Untitled".

Par défaut, l'interface utilisateur du *Qt Designer* consiste en plusieurs fenêtres de haut niveau. Si vous préférez une interface de style MDI, avec une fenêtre de haut niveau et plusieurs sous-fenêtres, cliquez sur Edit > User Interface Mode > Docked Window (voir Figure 2.5).

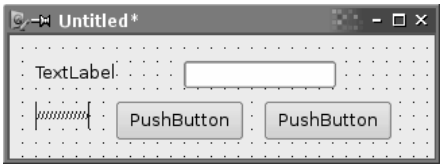
La première étape consiste à créer les widgets enfants et à les placer sur le formulaire. Créez une étiquette, un éditeur de lignes, un élément d'espacement horizontal et deux boutons de commande. Pour chaque élément, faites glisser son nom ou son icône depuis la boîte des widgets du *Qt Designer* vers son emplacement sur le formulaire. L'élément d'espacement, qui est invisible dans le formulaire final, est affiché dans le *Qt Designer* sous forme d'un ressort bleu.

Figure 2.5
Le Qt Designer en mode d'affichage fenêtres ancrées sous Windows



Faites glisser le bas du formulaire vers le haut pour le rétrécir. Vous devriez voir un formulaire similaire à celui de la Figure 2.6. Ne perdez pas trop de temps à positionner les éléments sur le formulaire ; les gestionnaires de disposition de Qt les disposeront précisément par la suite.

Figure 2.6
Le formulaire avec quelques widgets

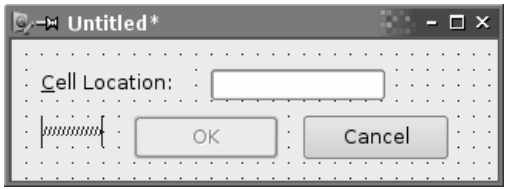


Configurez les propriétés de chaque widget à l'aide de l'éditeur de propriétés du *Qt Designer :*

1. Cliquez sur l'étiquette de texte. Assurez-vous que la propriété `objectName` est `label` et définissez la propriété `text` en `&Cell Location:`.
2. Cliquez sur l'éditeur de lignes. Vérifiez que la propriété `objectName` est `lineEdit`.
3. Cliquez sur le premier bouton. Configurez la propriété `objectName` en `okButton`, la propriété `enabled` en `false`, la propriété `text` en `OK` et la propriété `default` en `true`.
4. Cliquez sur le second bouton. Définissez la propriété `objectName` en `cancelButton` et la propriété `text` en `Cancel`.
5. Cliquez sur l'arrière-plan du formulaire pour sélectionner ce dernier. Définissez `objectName` en `GoToCellDialog` et `windowTitle` en `Go to Cell`.

Tous les widgets sont correctement présentés, sauf l'étiquette de texte, qui affiche &Cell Location. Cliquez sur Edit > Edit Buddies pour passer dans un mode spécial qui vous permet de configurer les compagnons. Cliquez ensuite sur l'étiquette et faites glisser la flèche rouge vers l'éditeur de lignes. L'étiquette devrait présenter le texte "Cell Location" et l'éditeur de lignes devrait être son widget compagnon. Cliquez sur Edit > Edit widgets pour quitter le mode des compagnons.

Figure 2.7
Le formulaire dont les propriétés sont définies



La prochaine étape consiste à disposer les widgets sur le formulaire :

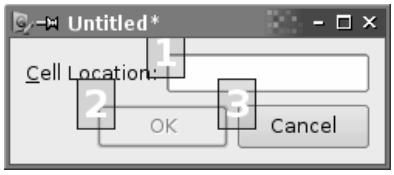
1. Cliquez sur l'étiquette Cell Location et maintenez la touche Maj enfoncée quand vous cliquez sur l'éditeur de lignes, de manière à ce qu'ils soient sélectionnés tous les deux. Cliquez sur Form > Lay Out Horizontally.
2. Cliquez sur l'élément d'espacement, maintenez la touche Maj enfoncée et appuyez sur les boutons OK et Cancel du formulaire. Cliquez sur Form > Lay Out Horizontally.
3. Cliquez sur l'arrière-plan du formulaire pour annuler toute sélection d'élément, puis cliquez sur Form > Lay Out Vertically.
4. Cliquez sur Form > Ajust Size pour redimensionner le formulaire.

Les lignes rouges qui apparaissent sur le formulaire montrent les dispositions qui ont été créées. Elles ne s'affichent pas lorsque le formulaire est exécuté.

Figure 2.8
Le formulaire avec les dispositions

Cliquez à présent sur Edit > Edit Tab Order. Un nombre apparaîtra dans un rectangle bleu à côté de chaque widget qui peut devenir actif (voir Figure 2.9). Cliquez sur chaque widget dans l'ordre dans lequel vous voulez qu'ils reçoivent le focus, puis cliquez sur Edit > Edit widgets pour quitter le mode d'édition de l'ordre de tabulation.

Figure 2.9
Définir l'ordre de tabulation du formulaire



Pour avoir un aperçu de la boîte de dialogue, sélectionnez l'option Form > Preview du menu. Vérifiez l'ordre de tabulation en appuyant plusieurs fois sur la touche Tab. Fermez la boîte de dialogue en appuyant sur le bouton de fermeture dans la barre de titre.

Enregistrez la boîte de dialogue sous `gotocelldialog.ui` dans un répertoire appelé `gotocell`, et créez un fichier `main.cpp` dans le même répertoire grâce à un éditeur de texte ordinaire :

```
#include <QApplication>
#include <QDialog>

#include "ui_gotocelldialog.h"

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);

   Ui::GoToCellDialog ui;
   QDialog *dialog = new QDialog;
   ui.setupUi(dialog);
   dialog->show();

   return app.exec();
}
```

Exécutez maintenant `qmake` pour créer un fichier .pro et un Makefile (`qmake -project`; `qmake goto-cell.pro`). L'outil `qmake` est suffisamment intelligent pour détecter le fichier de l'interface utilisateur `gotocelldialog.ui` et pour générer les règles appropriées du fichier Makefile. Il va donc appeler `uic`, le compilateur Qt de l'interface utilisateur. L'outil `uic` convertit `gotocelldialog.ui` en langage C++ et intègre les résultats dans `ui_gotocelldialog.h`.

Le fichier `ui_gotocelldialog.h` généré contient la définition de la classe `Ui::GoToCell-Dialog` qui est un équivalent C++ du fichier `gotocelldialog.ui`. La classe déclare des variables membres qui stockent les widgets enfants et les dispositions du formulaire, et une fonction `setupUi()` qui initialise le formulaire. Voici la syntaxe de la classe générée :

```
class Ui::GoToCellDialog
{
public:
   QLabel *label;
   QLineEdit *lineEdit;
   QSpacerItem *spacerItem;
   QPushButton *okButton;
   QPushButton *cancelButton;
   ...



   void setupUi(QWidget *widget) {
      ...
   }
};
```

Cette classe n'hérite pas de n'importe quelle classe Qt. Quand vous utilisez le formulaire dans `main.cpp`, vous créez un `QDialog` et vous le transmettez à `setupUi()`.

Si vous exécutez le programme maintenant, la boîte de dialogue fonctionnera, mais pas exactement comme vous le souhaitiez :

- Le bouton OK est toujours désactivé.
- Le bouton Cancel ne fait rien.
- L'éditeur de lignes accepte n'importe quel texte, au lieu d'accepter uniquement des emplacements de cellule valides.

Pour faire fonctionner correctement la boîte de dialogue, vous devrez écrire du code. La meilleure approche consiste à créer une nouvelle classe qui hérite à la fois de `QDialog` et de `Ui::GoToCellDialog` et qui implémente la fonctionnalité manquante (ce qui prouve que tout problème logiciel peut être résolu simplement en ajoutant une autre couche d'indirection). Notre convention de dénomination consiste à attribuer à cette nouvelle classe le même nom que la classe générée par `uic`, mais sans le préfixe `Ui::`.

A l'aide d'un éditeur de texte, créez un fichier nommé `gotocelldialog.h` qui contient le code suivant :

```
#ifndef GOTOCELLDIALOG_H
#define GOTOCELLDIALOG_H

#include <QDialog>

#include "ui_gotocelldialog.h"

class GoToCellDialog : public QDialog, public Ui::GoToCellDialog
{
   Q_OBJECT

public:
   GoToCellDialog(QWidget *parent = 0);

private slots:
   void on_lineEdit_textChanged();
};

#endif
```

L'implémentation fait partie de `gotocelldialog.cpp` :

```
#include <QtGui>

#include "gotocelldialog.h"

GoToCellDialog::GoToCellDialog(QWidget *parent)
   : QDialog(parent)
{
   setupUi(this);

   QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
```

```
    lineEdit->setValidator(new QRegExpValidator(regExp, this));

    connect(okButton, SIGNAL(clicked()), this, SLOT(accept()));
    connect(cancelButton, SIGNAL(clicked()), this, SLOT(reject()));
}

void GoToCellDialog::on_lineEdit_textChanged()
{
    okButton->setEnabled(lineEdit->hasAcceptableInput());
}
```

Dans le constructeur, nous appelons `setupUi()` pour initialiser le formulaire. Grâce à l'héritage multiple, nous pouvons accéder directement aux membres de `Ui::GoToCellDialog`. Après avoir créé l'interface utilisateur, `setupUi()` connectera également automatiquement tout slot qui respecte la convention de dénomination `on_Nomobjet_NomSignal()` au signal `nomSignal()` correspondant de `objectName`. Dans notre exemple, cela signifie que `setupUi()` établira la connexion signal-slot suivante :

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SLOT(on_lineEdit_textChanged()));
```

Toujours dans le constructeur, nous définissons un validateur qui restreint la plage des valeurs en entrée. Qt propose trois classes de validateurs :`QIntValidator`, `QDoubleValidator` et `QRegExpValidator`. Nous utilisons ici `QRegExpValidator` avec l'expression régulière "[A-Za-z][1-9][0-9]{0,2}", qui signifie : autoriser une lettre majuscule ou minuscule, suivie d'un chiffre entre 1 et 9, puis de zéro, un ou deux chiffres entre 0 et 9. (En guise d'introduction aux expressions régulières, consultez la documentation de la classe `QRegExp`.)

Si vous transmettez ceci au constructeur de `QRegExpValidator`, vous en faites un enfant de l'objet `GoToCellDialog`. Ainsi, vous n'avez pas besoin de prévoir la suppression de `QRegExpValidator` ; il sera supprimé automatiquement en même temps que son parent.

Le mécanisme parent-enfant de Qt est implémenté dans `QObject`. Quand vous créez un objet (un widget, un validateur, ou autre) avec un parent, le parent ajoute l'objet à sa liste d'enfants. Quand le parent est supprimé, il parcourt sa liste d'enfants et les supprime. Les enfants eux-mêmes effacent ensuite tous leurs enfants, et ainsi de suite jusqu'à ce qu'il n'en reste plus aucun.

Ce mécanisme parent-enfant simplifie nettement la gestion de la mémoire, réduisant les risques de fuites de mémoire. Les seuls objets que vous devrez supprimer explicitement sont les objets que vous créez avec `new` et qui n'ont pas de parent. Et si vous supprimez un objet enfant avant son parent, Qt supprimera automatiquement cet objet de la liste des enfants du parent.

S'agissant des widgets, le parent a une signification supplémentaire : les widgets enfants sont affichés dans la zone du parent. Quand vous supprimez le widget parent, l'enfant est effacé de la mémoire mais également de l'écran.

A la fin du constructeur, nous connectons le bouton OK au slot `accept()` de `QDialog` et le bouton Cancel au slot `reject()`. Les deux slots ferment la boîte de dialogue, mais `accept()` définit la valeur de résultat de la boîte de dialogue en `QDialog::Accepted` (qui est égal à 1),

et `reject()` configure la valeur en `QDialog::Rejected` (égal à 0). Quand nous utilisons cette boîte de dialogue, nous avons la possibilité d'utiliser la valeur de résultat pour voir si l'utilisateur a cliqué sur OK et agir de façon appropriée.

Le slot `on_lineEdit_textChanged()` active ou désactive le bouton OK, selon que l'éditeur de lignes contient un emplacement de cellule valide ou non. `QLineEdit::hasAcceptableInput()` emploie le validateur que nous avons défini dans le constructeur.

Ceci termine la boîte de dialogue. Vous pouvez désormais réécrire `main.cpp` pour l'utiliser :

```
#include <QApplication>

#include "gotocelldialog.h"

int main(int argc, char *argv[])
{

   QApplication app(argc, argv);
   GoToCellDialog *dialog = new GoToCellDialog;
   dialog->show();
   return app.exec();
}
```

Régénérez l'application (`qmake -project ; qmake gotocell.pro`) et exécutez-la à nouveau. Tapez "A12" dans l'éditeur de lignes et vous verrez que le bouton OK s'active. Essayez de saisir du texte aléatoire pour vérifier que le validateur effectue bien sa tâche. Cliquez sur Cancel pour fermer la boîte de dialogue.

L'un des avantages du *Qt Designer*, c'est que les programmeurs sont libres de modifier la conception de leurs formulaires sans être obligés de changer leur code source. Quand vous développez un formulaire simplement en rédigeant du code C++, les modifications apportées à la conception peuvent vous faire perdre énormément de temps. Grâce au *Qt Designer*, vous gagnez en efficacité parce que `uic` régénère simplement le code source pour tout formulaire modifié. L'interface utilisateur de la boîte de dialogue est enregistrée dans un fichier `.ui` (un format de fichier basé sur le langage XML), alors que la fonctionnalité personnalisée est implémentée en sous-classant la classe générée par `uic`.

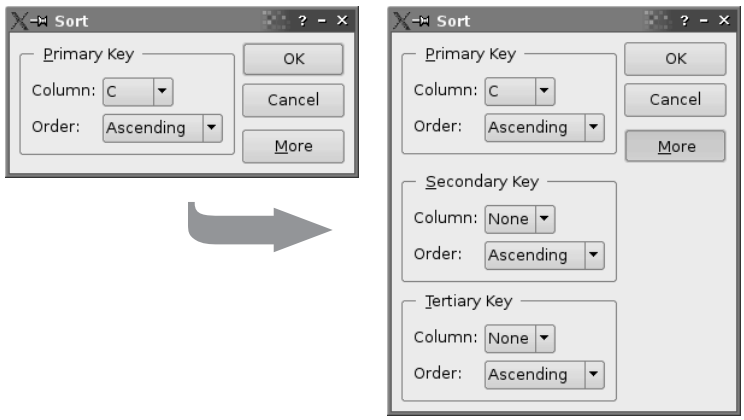
Boîtes de dialogue multiformes

Nous avons vu comment créer des boîtes de dialogue qui affichent toujours les mêmes widgets lors de leur utilisation. Dans certains cas, il est souhaitable de proposer des boîtes de dialogue dont la forme peut varier. Les deux types les plus courants de boîtes de dialogue multiformes sont les *boîtes de dialogue extensibles* et les *boîtes de dialogue multipages*. Ces deux types de boîtes de dialogue peuvent être implémentés dans Qt, simplement dans du code ou par le biais du *Qt Designer*.

Les boîtes de dialogue extensibles ont habituellement une apparence simple, mais elles proposent un bouton de basculement qui permet à l'utilisateur d'alterner entre les apparences simple et développée de la boîte de dialogue. Ces boîtes de dialogue sont généralement utilisées dans des applications qui tentent de répondre à la fois aux besoins des utilisateurs occasionnels et à ceux des utilisateurs expérimentés, masquant les options avancées à moins que l'utilisateur ne demande explicitement à les voir. Dans cette section, nous utiliserons le *Qt Designer* afin de créer la boîte de dialogue extensible présentée en Figure 2.10.

C'est une boîte de dialogue Sort dans un tableur, où l'utilisateur a la possibilité de sélectionner une ou plusieurs colonnes à trier. L'apparence simple de cette boîte de dialogue permet à l'utilisateur de saisir une seule clé de tri, et son apparence développée propose deux clés de tri supplémentaires. Grâce au bouton More, l'utilisateur bascule entre les apparences simple et développée.

Figure 2.10
La boîte de dialogue Sort dans ses deux versions, simple et développée

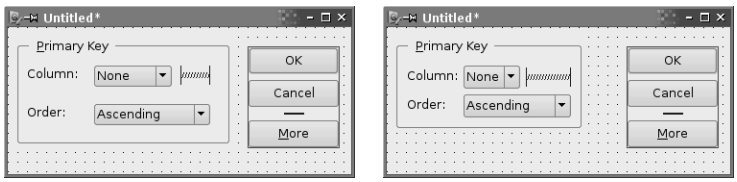


Nous créerons le widget avec son apparence développée dans le *Qt Designer*, et nous masquerons les clés secondaires et tertiaires à l'exécution. Le widget semble complexe, mais il est assez simple à réaliser dans le *Qt Designer*. L'astuce est de se charger d'abord de la clé primaire, puis de la dupliquer deux fois pour obtenir les clés secondaires et tertiaires :

1. Cliquez sur File > New Form et sélectionnez le modèle "Dialog with Buttons Right".
2. Créez le bouton More et faites-le glisser dans la disposition verticale, sous l'élément d'espacement vertical. Définissez la propriété `text` du bouton More en `&More` et sa propriété `checkable` en `true`. Configurez la propriété `default` du bouton OK en `true`.
3. Créez une zone de groupe, deux étiquettes, deux zones de liste déroulante et un élément d'espacement horizontal, puis placez-les sur le formulaire.
4. Faites glisser le coin inférieur droit de la zone de groupe pour l'agrandir. Puis déplacez les autres widgets dans la zone de groupe pour les positionner à peu près comme dans la Figure 2.11 (a).

5. Faites glisser le bord droit de la seconde zone de liste déroulante, de sorte qu'elle soit environ deux fois plus grande que la première zone de liste.
6. Définissez la propriété `title` de la zone de groupe en `&Primary Key`, la propriété `text` de la première étiquette en `Column:` et celle de la deuxième étiquette en `Order:`.
7. Cliquez du bouton droit sur la première zone de liste déroulante et sélectionnez Edit Items dans le menu contextuel pour ouvrir l'éditeur de zone de liste déroulante du *Qt Designer*. Créez un élément avec le texte "None".
8. Cliquez du bouton droit sur la seconde zone de liste déroulante et sélectionnez Edit Items. Créez les éléments "Ascending" et "Descending".
9. Cliquez sur la zone de groupe, puis sur Form > Lay Out in a Grid. Cliquez à nouveau sur la zone de groupe et sur Form > Adjust Size. Vous aboutirez à la disposition affichée dans la Figure 2.11 (b).

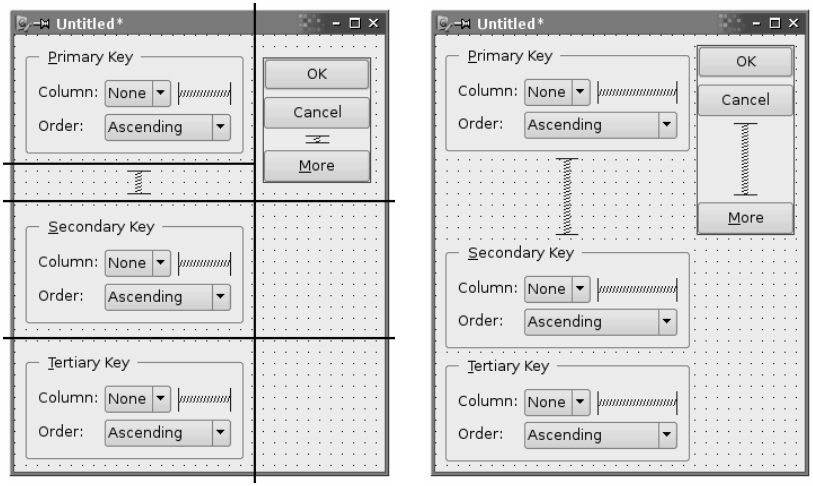
Figure 2.11
Disposer les enfants de la zone de groupe dans une grille



(a) Sans disposition (b) Avec disposition

Si une disposition ne s'avère pas très bonne ou si vous avez fait une erreur, vous pouvez toujours cliquer sur Edit > Undo ou Form > Break Layout, puis repositionner les widgets et réessayer.

Figure 2.12
Disposer les enfants du formulaire dans une grille



(a) Sans disposition (b) Avec disposition

Nous allons maintenant ajouter les zones de groupe Secondary Key et Tertiary Key :

1. Prenez garde à ce que la fenêtre soit assez grande pour accueillir les composants supplémentaires.
2. Maintenez la touche Ctrl enfoncée (Alt sur Mac) et cliquez sur la zone de groupe Primary Key pour en créer une copie (et de son contenu) au-dessus de l'original. Faites glisser la copie sous la zone de groupe originale en gardant toujours la touche Ctrl (ou Alt) enfoncée. Répétez ce processus pour créer une troisième zone de groupe, en la faisant glisser sous la deuxième zone.
3. Transformez leurs propriétés `title` en `&Secondary Key` et `&Tertiary Key`.
4. Créez un élément d'espacement vertical et placez-le entre la zone de la clé primaire et celle de la clé secondaire.
5. Disposez les widgets comme illustré en Figure 2.12 (a).
6. Cliquez sur le formulaire pour annuler la sélection de tout widget, puis sur Form > Lay Out in a Grid. Le formulaire devrait désormais correspondre à celui de la Figure 2.12 (b).
7. Définissez la propriété `sizeHint` des deux éléments d'espacement verticaux en [20, 0].

La disposition de type grille qui en résulte comporte deux colonnes et quatre lignes, ce qui fait un total de huit cellules. La zone de groupe Primary Key, l'élément d'espacement vertical le plus à gauche, les zones de groupe Secondary Key et Tertiary Key occupent chacun une seule cellule. La disposition verticale qui contient les boutons OK, Cancel et More occupe deux cellules. Il reste donc deux cellules vides en bas à droite de la boîte de dialogue. Si ce n'est pas le cas, annulez la disposition, repositionnez les widgets et essayez à nouveau.

Renommez le formulaire "SortDialog" et changez le titre de la fenêtre en "Sort".Définissez les noms des widgets enfants comme dans la Figure 2.13.

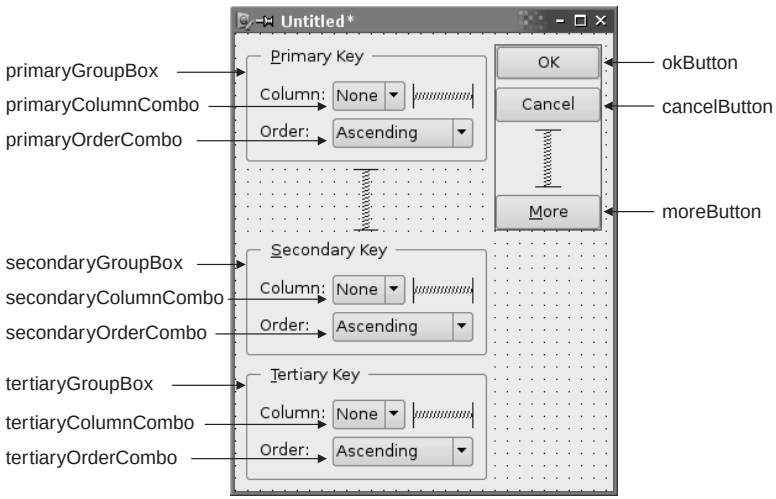


Figure 2.13
Nommer les widgets du formulaire

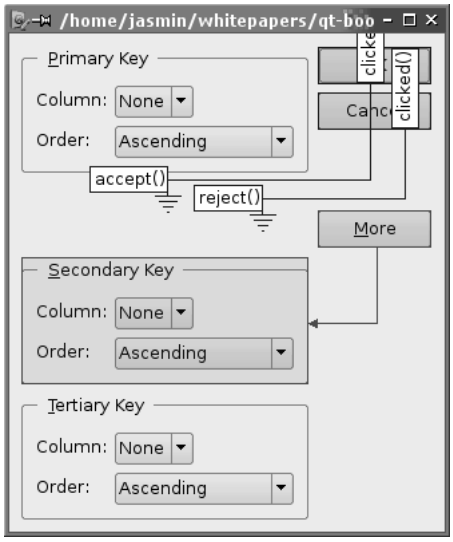
Cliquez sur Edit > Edit Tab Order. Cliquez sur chaque zone de liste déroulante de haut en bas, puis cliquez sur les boutons OK, Cancel et More situés à droite. Cliquez sur Edit > Edit Widgets pour quitter le mode édition de l'ordre de tabulation.

Maintenant que le formulaire a été conçu, nous sommes prêts à le rendre fonctionnel en configurant certaines connexions signal-slot. Le *Qt Designer* vous permet d'établir des connexions entre les widgets qui font partie du même formulaire. Nous devons établir deux connexions.

Cliquez sur Edit > Edit Signals/Slots pour passer en mode de connexion dans le *Qt Designer.* Les connexions sont représentées par des flèches bleues entre les widgets du formulaire. Vu que nous avons choisi le modèle "Dialog with Buttons Right", les boutons OK et Cancel sont déjà connectés aux slots `accept()` et `reject()` de `QDialog`. Les connexions sont également répertoriées dans l'éditeur de signal/slot du *Qt Designer.*

Pour établir une connexion entre deux widgets, cliquez sur le widget "expéditeur" et faites glisser la flèche rouge vers le widget "destinataire".Une boîte de dialogue s'ouvre où vous pouvez choisir le signal et le slot à connecter.

Figure 2.14
Connecter les widgets du formulaire



La première connexion est à établir entre `moreButton` et `secondaryGroupBox`. Faites glisser la flèche rouge entre ces deux widgets, puis choisissez `toggled(bool)` comme signal et `setVisible(bool)` comme slot. Par défaut, le *Qt Designer* ne répertorie pas `setVisible(bool)` dans sa liste de slots, mais il apparaîtra si vous activez l'option Show all signals and slots.

Vous devez ensuite créer une connexion entre le signal `toggled(bool)` de `moreButton` et le slot `setVisible(bool)` de `tertiaryGroupBox`. Lorsque les connexions sont effectuées, cliquez sur Edit > Edit Widgets pour quitter le mode de connexion.

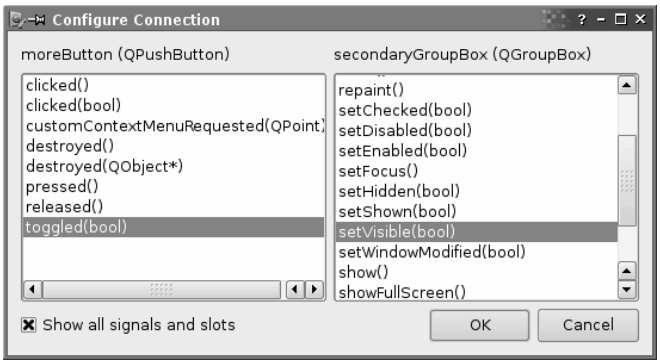


Figure 2.15
L'éditeur de connexions du Qt Designer

Enregistrez la boîte de dialogue sous `sortdialog.ui` dans un répertoire appelé `sort`. Pour ajouter du code au formulaire, vous emploierez la même technique d'héritage multiple que celle utilisée pour la boîte de dialogue Go-to-Cell de la section précédente.

Créez tout d'abord un fichier `sortdialog.h` qui comporte les éléments suivants :

```
#ifndef SORTDIALOG_H
#define SORTDIALOG_H

#include <QDialog>

#include "ui_sortdialog.h"

class SortDialog : public QDialog, public Ui::SortDialog
{
   Q_OBJECT

public:
   SortDialog(QWidget *parent = 0);

   void setColumnRange(QChar first, QChar last);
};

#endif
```

Puis créez `sortdialog.cpp` :

```
1  #include <QtGui>

2  #include "sortdialog.h"

3  SortDialog::SortDialog(QWidget *parent)
4      : QDialog(parent)
5  {
6      setupUi(this);

7      secondaryGroupBox->hide();
8      tertiaryGroupBox->hide();
```

```
 9      layout()->setSizeConstraint(QLayout::SetFixedSize);

10      setColumnRange('A', 'Z');
11  }

12  void SortDialog::setColumnRange(QChar first, QChar last)
13  {
14      primaryColumnCombo->clear();
15      secondaryColumnCombo->clear();
16      tertiaryColumnCombo->clear();

17      secondaryColumnCombo->addItem(tr("None"));
18      tertiaryColumnCombo->addItem(tr("None"));

19      primaryColumnCombo->setMinimumSize(
20          secondaryColumnCombo->sizeHint());
21      QChar ch = first;
22      while (ch <= last) {
23          primaryColumnCombo->addItem(QString(ch));
24          secondaryColumnCombo->addItem(QString(ch));
25          tertiaryColumnCombo->addItem(QString(ch));
26          ch = ch.unicode() + 1;
27      }
28  }
```

Le constructeur masque les zones secondaire et tertiaire de la boîte de dialogue. Il définit aussi la propriété `sizeConstraint` de la disposition du formulaire en `QLayout::SetFixedSize`, l'utilisateur ne pourra donc pas la redimensionner. La disposition se charge ensuite de redimensionner automatiquement la boîte de dialogue quand des widgets enfants sont affichés ou masqués, vous être donc sûr que la boîte de dialogue sera toujours présentée dans sa taille optimale.

Le slot `setColumnRange()` initialise le contenu des zones de liste déroulante en fonction des colonnes sélectionnées dans le tableur. Nous insérons un élément "None" dans ces zones de liste pour les clés secondaire et tertiaire (facultatives).

Les lignes 19 et 20 présentent un comportement subtil de la disposition. La fonction `QWidget::sizeHint()` retourne la taille "idéale" d'un widget, ce que le système de disposition essaie de respecter. Ceci explique pourquoi les différents types de widgets, ou des widgets similaires avec un contenu différent, peuvent se voir attribuer des tailles différentes par le système de disposition. Concernant les zones de liste déroulante, cela signifie que les zones secondaire et tertiaire qui contiennent "None" seront plus grandes que la zone primaire qui ne contient que des entrées à une lettre. Pour éviter cette incohérence, nous définissons la taille minimale de la zone de liste déroulante primaire en taille idéale de la zone *secondaire*.

Voici une fonction de test `main()` qui configure la plage de manière à inclure les colonnes C à F, puis affiche la boîte de dialogue :

```
#include <QApplication>

#include "sortdialog.h"
```

```
int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   SortDialog *dialog = new SortDialog;
   dialog->setColumnRange('C', 'F');
   dialog->show();
   return app.exec();
}
```

Ceci termine la boîte de dialogue extensible. Vous pouvez constater que ce type de boîte de dialogue n'est pas plus compliqué à concevoir qu'une boîte de dialogue ordinaire : tout ce dont vous avez besoin, c'est un bouton de basculement, quelques connexions signal-slot supplémentaires et une disposition non redimensionnable. Dans des applications de production, il est assez fréquent que le bouton qui contrôle l'extension affiche le texte Advanced >>> quand seule la boîte de dialogue de base est visible et Advanced <<< quand elle est développée. C'est facile à concevoir dans Qt en appelant `setText()` sur `QPushButton` dès qu'on clique dessus.

L'autre type courant de boîte de dialogue multiforme, les boîtes de dialogue multipages, est encore plus facile à concevoir dans Qt, soit en créant le code, soit par le biais du *Qt Designer*. De telles boîtes de dialogue peuvent être générées de diverses manières.

- Un `QTabWidget` peut être exploité indépendamment. Il propose une barre d'onglets en haut qui contrôle un `QStackedWidget` intégré.
- Un `QListWidget` et un `QStackedWidget` peuvent être employés ensemble, avec l'élément en cours de `QListWidget` qui détermine quelle page est affichée par `QStackedWidget`, en connectant le signal `QListWidget::currentRowChanged()` au slot `QStackedWidget::setCurrentIndex()`.
- Un `QTreeWidget` peut être utilisé avec un `QStackedWidget` de la même façon qu'avec un `QListWidget`.
- La classe `QStackedWidget` est abordée au Chapitre 6.

Boîtes de dialogue dynamiques

Les boîtes de dialogue dynamiques sont créées depuis les fichiers `.ui` du *Qt Designer* au moment de l'exécution. Au lieu de convertir le fichier `.ui` en code C++ grâce à `uic`, vous pouvez charger le fichier à l'exécution à l'aide de la classe `QUiLoader` :

```
QUiLoader uiLoader;
QFile file("sortdialog.ui");
QWidget *sortDialog = uiLoader.load(&file);
if (sortDialog) {
   ...
}
```

Vous pouvez accéder aux widgets enfants du formulaire en utilisant `QObject::findChild<T>()` :

```
QComboBox *primaryColumnCombo =
      sortDialog->findChild<QComboBox *>("primaryColumnCombo");
if (primaryColumnCombo) {
   ...
}
```

La fonction `findChild<T>()` est une fonction membre modèle qui retourne l'objet enfant qui correspond au nom et au type donné. Vu les limites du compilateur, elle n'est pas disponible pour MSVC 6. Si vous devez utiliser le compilateur MSVC 6, appelez plutôt la fonction globale `qFindChild<T>()` qui fonctionne exactement de la même façon.

La classe `QUiLoader` se situe dans une bibliothèque à part. Pour employer `QUiLoader` depuis une application Qt, vous devez ajouter cette ligne de code au fichier `.pro` de l'application :

```
CONFIG += uitools
```

Les boîtes de dialogue dynamiques vous permettent de modifier la disposition d'un formulaire sans recompiler l'application. Elles peuvent aussi servir à créer des applications client léger, où l'exécutable intègre principalement un formulaire frontal et où tous les autres formulaires sont créés comme nécessaire.

Classes de widgets et de boîtes de dialogue intégrées

Qt propose un ensemble complet de widgets intégrés et de boîtes de dialogue courantes adaptés à la plupart des situations. Dans cette section, nous allons présenter une capture de la plupart d'entre eux. Quelques widgets spécialisés ne sont étudiés qu'ultérieurement : les widgets de fenêtre principale comme `QMenuBar`, `QToolBar` et `QStatusBar` sont abordés dans le Chapitre 3, et les widgets liés à la disposition, tels que `QSplitter` et `QScrollArea`, sont analysés dans le Chapitre 6. La majorité des widgets intégrés et des boîtes de dialogue est présentée dans les exemples de ce livre. Dans les captures suivantes, les widgets sont affichés avec le style Plastique.

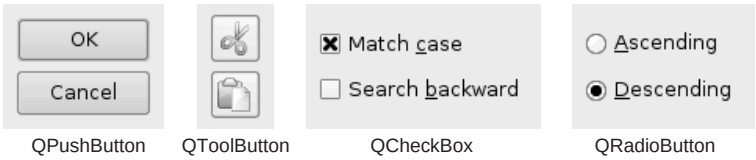
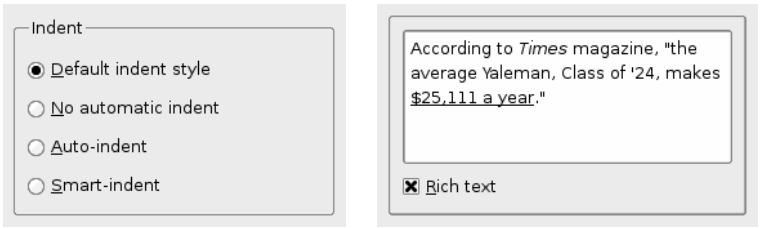


Figure 2.16
Les widgets bouton de Qt

Qt propose quatre types de "boutons" : `QPushButton`, `QToolButton`, `QCheckBox` et `QRadioButton`. `QPushButton` et `QToolButton` sont le plus souvent ajoutés pour initier une action quand on clique dessus, mais ils peuvent aussi se comporter comme des boutons de basculement

(clic pour enfoncer, clic pour restaurer). `QCheckBox` peut servir pour les options indépendantes on/off, alors que les `QRadioButton` s'excluent mutuellement.

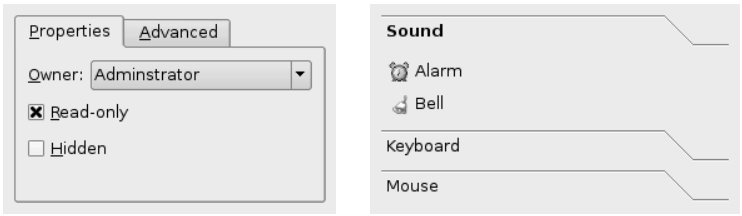
Figure 2.17
Les widgets conteneurs à une seule page de Qt



QGroupBox QFrame

Les widgets conteneurs de Qt sont des widgets qui contiennent d'autres widgets. `QFrame` peut aussi être employé seul pour tracer simplement des lignes et il est hérité par la plupart des autres classes de widgets, dont `QToolBox` et `QLabel`.

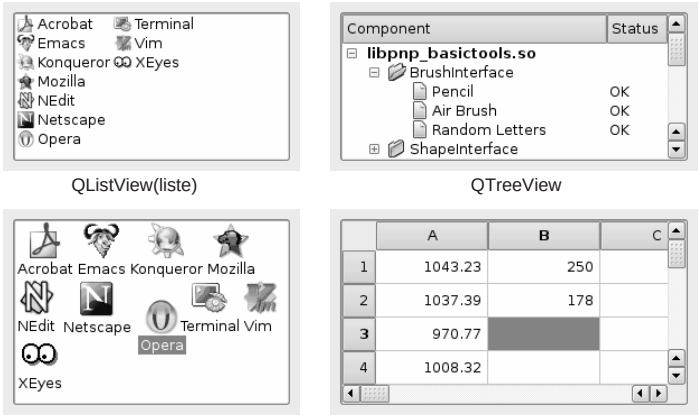
Figure 2.18
Les widgets conteneurs multipages de Qt



QTabWidget QToolBox

`QTabWidget` et `QToolBox` sont des widgets multipages. Chaque page est un widget enfant, et les pages sont numérotées en commençant à 0.

Figure 2.19
Les widgets d'affichage d'éléments de Qt



QListView(liste) QTreeView

QListView(icônes) QTableView

Les affichages d'éléments sont optimisés pour gérer de grandes quantités de données et font souvent appel à des barres de défilement. Le mécanisme de la barre de défilement est implémenté dans `QAbstractScrollArea`, une classe de base pour les affichages d'éléments et d'autres types de widgets à défilement.

Qt fournit quelques widgets simplement destinés à la présentation des informations. `QLabel` est le plus important d'entre eux et peut être employé pour afficher un texte enrichi (grâce à une syntaxe simple de style HTML) et des images.

`QTextBrowser` est une sous-classe de `QTextEdit` en lecture seule qui prend en charge la syntaxe HTML de base, y compris les listes, les tables, les images et les liens hypertexte. L'*Assistant* de Qt se sert de `QTextBrowser` pour présenter la documentation à l'utilisateur.

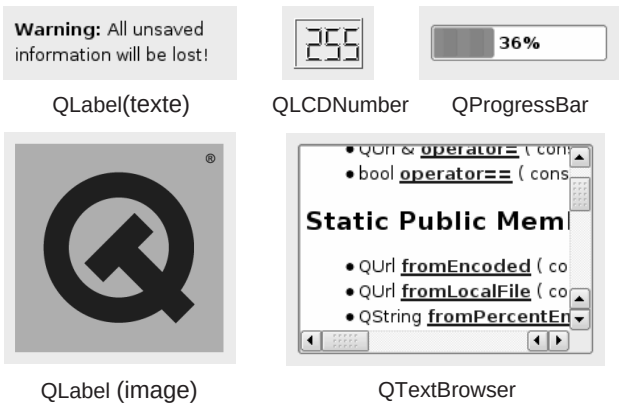


Figure 2.20
Les widgets d'affichage de Qt

Qt propose plusieurs widgets pour les entrées de données. `QLineEdit` peut contrôler son entrée par le biais d'un masque de saisie ou d'un validateur. `QTextEdit` est une sous-classe de `QAbstractScrollArea` capable de modifier de grandes quantités de texte.

Qt met à votre disposition un ensemble standard de boîtes de dialogue courantes pratiques pour demander à l'utilisateur de choisir une couleur, une police ou un fichier ou d'imprimer un document.

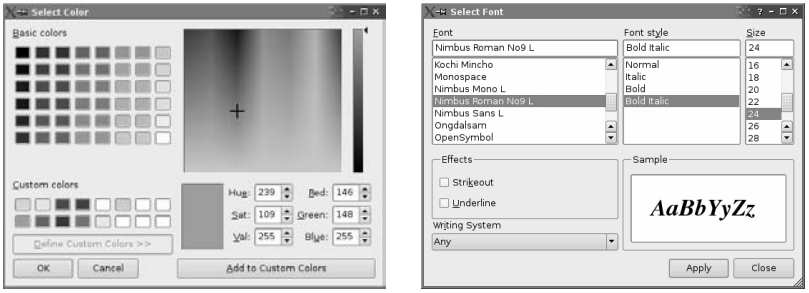
Sous Windows et Mac OS X, Qt exploite les boîtes de dialogue natives plutôt que ses propres boîtes de dialogue si possible.

Une boîte de message polyvalente et une boîte de dialogue d'erreur qui conserve les messages affichés apparaissent. La progression des opérations longues peut être indiquée dans un `QProgressDialog` ou `QProgressBar` présenté précédemment. `QInputDialog` se révèle très pratique quand une seule ligne de texte ou un seul chiffre est demandé à l'utilisateur.

Les widgets intégrés et les boîtes de dialogue courantes mettent à votre disposition de nombreuses fonctionnalités prêtes à l'emploi. Des exigences plus particulières peuvent souvent être satisfaites en configurant les propriétés du widget ou en connectant des signaux à des slots et en implémentant un comportement personnalisé dans les slots.

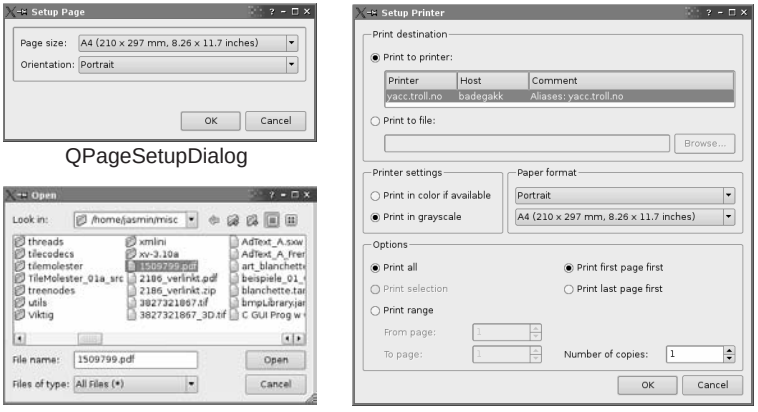
Figure 2.21
Les widgets d'entrée de Qt

Figure 2.22
Les boîtes de dialogue relatives à la couleur et à la police de Qt



QColorDialog QFontDialog

Figure 2.23
Les boîtes de dialogue relatives à l'impression et aux fichiers de Qt



QPageSetupDialog

QFileDialog QPrintDialog

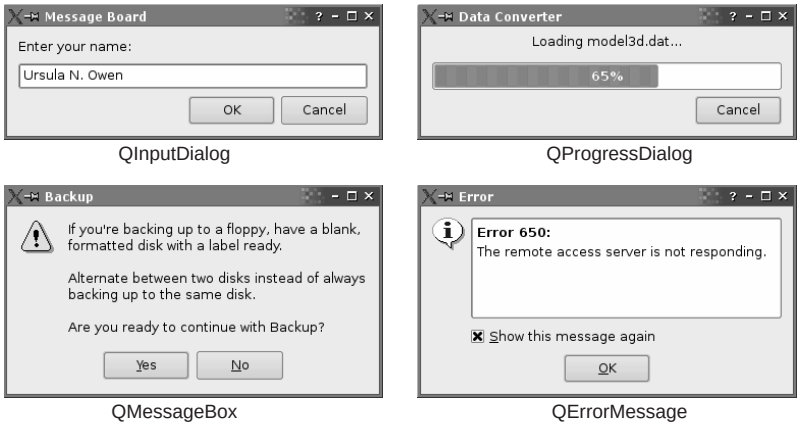


Figure 2.24
Les boîtes de dialogue de feedback de Qt

Dans certains cas, il est judicieux de créer un widget personnalisé en partant de zéro. Qt facilite énormément ce processus, et les widgets personnalisés peuvent accéder à la même fonction de dessin indépendante de la plate-forme que les widgets intégrés de Qt. Les widgets personnalisés peuvent même être intégrés par le biais du *Qt Designer*, de sorte qu'ils puissent être employés de la même façon que les widgets intégrés de Qt. Le Chapitre 5 vous explique comment créer des widgets personnalisés.

3

Créer des fenêtres principales

Au sommaire de ce chapitre

- ✔ Dérivation de QMainWindow
- ✔ Créer des menus et des barres d'outils
- ✔ Configurer la barre d'état
- ✔ Implémenter le menu File
- ✔ Utiliser des boîtes de dialogue
- ✔ Stocker des paramètres
- ✔ Documents multiples
- ✔ Pages d'accueil

Ce chapitre vous apprendra à créer des fenêtres principales avec Qt. Vous serez ainsi capable de concevoir toute l'interface utilisateur d'une application, constituée de menus, de barres d'outils, d'une barre d'état et d'autant de boîtes de dialogue que nécessaire.

La fenêtre principale d'une application fournit le cadre dans lequel l'interface utilisateur est générée. Celle de l'application Spreadsheet illustrée en Figure 3.1 servira de base pour l'étude dans ce chapitre. Cette application emploie les boîtes de dialogue Find, Go-to-Cell et Sort créées au Chapitre 2.

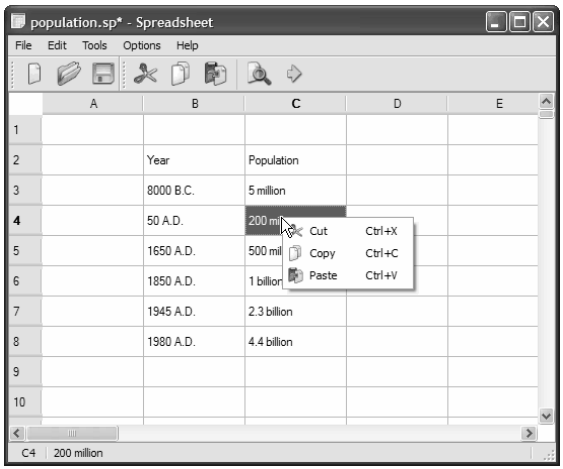


Figure 3.1
L'application Spreadsheet

Derrière la plupart des applications GUI se cache du code qui fournit les fonctionnalités sous-jacentes – par exemple, le code qui lit et écrit des fichiers ou qui traite les données présentées dans l'interface utilisateur. Au Chapitre 4, vous verrez comment implémenter de telles fonctionnalités, toujours en utilisant l'application Spreadsheet comme exemple.

Dérivation de QMainWindow

La fenêtre principale d'une application est créée en dérivant `QMainWindow`. La plupart des techniques étudiées dans le Chapitre 2 pour créer des boîtes de dialogue s'appliquent également à la conception de fenêtres principales, puisque `QDialog` et `QMainWindow` héritent de `QWidget`.

Les fenêtres principales peuvent être créées à l'aide du *Qt Designer*, mais dans ce chapitre nous effectuerons tout le processus dans du code pour vous montrer le fonctionnement. Si vous préférez une approche plus visuelle, consultez le chapitre "Creating Main Windows in *Qt Designer*" dans le manuel en ligne de cet outil.

Le code source de la fenêtre principale de l'application Spreadsheet est réparti entre `mainwindow.h` et `mainwindow.cpp`. Commençons par examiner le fichier d'en-tête :

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class QAction;
class QLabel;
class FindDialog;
class Spreadsheet;
```

```
class MainWindow : public QMainWindow
{
   Q_OBJECT

public:
   MainWindow();

protected:
   void closeEvent(QCloseEvent *event);
```

Nous définissons la classe `MainWindow` comme une sous-classe de `QMainWindow`. Elle contient la macro `Q_ OBJECT` puisqu'elle fournit ses propres signaux et slots.

La fonction `closeEvent()` est une fonction virtuelle dans `QWidget` qui est appelée automatiquement quand l'utilisateur ferme la fenêtre. Elle est réimplémentée dans `MainWindow`, de sorte que vous puissiez poser à l'utilisateur la question standard "Voulez-vous enregistrer vos modifications ?" et sauvegarder les préférences de l'utilisateur sur le disque.

```
private slots:
    void newFile();
    void open();
    bool save();
    bool saveAs();
    void find();
    void goToCell();
    void sort();
    void about();
```

Certaines options de menu, telles que File > New (Fichier > Nouveau) et Help > About (Aide > A propos), sont implémentées comme des slots privés dans `MainWindow`. La majorité des slots ont une valeur de retour `void`, mais `save()` et `saveAs()` retournent une valeur `bool`. La valeur de retour est ignorée quand un slot est exécuté en réponse à un signal, mais lorsque vous invoquez un slot comme une fonction, la valeur de retour est disponible, comme si vous aviez appelé n'importe quelle fonction C++ ordinaire.

```
    void openRecentFile();
    void updateStatusBar();
    void spreadsheetModified();

private:
    void createActions();
    void createMenus();
    void createContextMenu();
    void createToolBars();
    void createStatusBar();
    void readSettings();
    void writeSettings();
    bool okToContinue();
    bool loadFile(const QString &fileName);
    bool saveFile(const QString &fileName);
```

```
    void setCurrentFile(const QString &fileName);
    void updateRecentFileActions();
    QString strippedName(const QString &fullFileName);
```

La fenêtre principale a besoin de slots privés et de plusieurs fonctions privées pour prendre en charge l'interface utilisateur.

```
    Spreadsheet *spreadsheet;
    FindDialog *findDialog;
    QLabel *locationLabel;
    QLabel *formulaLabel;
    QStringList recentFiles;
    QString curFile;

    enum { MaxRecentFiles = 5 };
    QAction *recentFileActions[MaxRecentFiles];
    QAction *separatorAction;

    QMenu *fileMenu;
    QMenu *editMenu;
    ...
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QAction *newAction;
    QAction *openAction;
    ...
    QAction *aboutQtAction;
};

#endif
```

En plus de ses slots et fonctions privés, `MainWindow` possède aussi de nombreuses variables privées. Elles seront analysées au fur et à mesure que vous les rencontrerez.

Nous allons désormais passer en revue l'implémentation :

```
#include <QtGui>
#include "finddialog.h"
#include "gotocelldialog.h"
#include "mainwindow.h"
#include "sortdialog.h"
#include "spreadsheet.h"
```

Nous incluons le fichier d'en-tête `<QtGui>`, qui contient la définition de toutes les classes Qt utilisées dans notre sous-classe. Nous englobons aussi certains fichiers d'en-tête personnalisés, notamment `finddialog.h`, `gotocelldialog.h` et `sortdialog.h` du Chapitre 2.

```
MainWindow::MainWindow()
{
    spreadsheet = new Spreadsheet;
    setCentralWidget(spreadsheet);

    createActions();
```

```
    createMenus();
    createContextMenu();
    createToolBars();
    createStatusBar();

    readSettings();

    findDialog = 0;

    setWindowIcon(QIcon(":/images/icon.png"));
    setCurrentFile("");
}
```

Dans le constructeur, nous commençons par créer un widget `Spreadsheet` et nous le configurons de manière à ce qu'il devienne le widget central de la fenêtre principale. Le widget central se trouve au milieu de la fenêtre principale (voir Figure 3.2). La classe `Spreadsheet` est une sous-classe de `QTableWidget` avec certaines fonctions de tableur, comme la prise en charge des formules de tableur. Nous l'implémenterons dans le Chapitre 4.

Nous appelons les fonctions privées `createActions()`, `createMenus()`, `createContextMenu()`, `createToolBars()` et `createStatusBar()` pour configurer le reste de la fenêtre principale. Nous invoquons également la fonction privée `readSettings()` afin de lire les paramètres stockés de l'application.

Nous initialisons le pointeur `findDialog` pour que ce soit un pointeur nul ; au premier appel de `MainWindow::find()`, nous créerons l'objet `FindDialog`.

A la fin du constructeur, nous définissons l'icône de la fenêtre en `icon.png`, un fichier PNG. Qt supporte de nombreux formats d'image, dont BMP, GIF[1], JPEG, PNG, PNM, XBM et XPM. L'appel de `QWidget::setWindowIcon()` définit l'icône affichée dans le coin supérieur gauche de la fenêtre. Malheureusement, il n'existe aucun moyen indépendant de la plate-forme pour configurer l'icône de l'application qui apparaît sur le bureau. Les procédures spécifiques à la plate-forme sont expliquées à l'adresse suivante :
http://doc.trolltech.com/4.1/appicon.html.

Les applications GUI utilisent généralement beaucoup d'images. Il existe plusieurs méthodes pour introduire des images dans une application. Les plus communes sont les suivantes :

- stocker des images dans des fichiers et les charger à l'exécution ;
- inclure des fichiers XPM dans le code source (Cela fonctionne parce que les fichiers XPM sont aussi des fichiers C++ valides.) ;
- utiliser le mécanisme des ressources de Qt.

1. La prise en charge du format GIF est désactivée dans Qt par défaut, parce que l'algorithme de décompression utilisé par les fichiers GIF était breveté dans certains pays où les brevets logiciels étaient reconnus. Nous pensons que ce brevet est arrivé à expiration dans le monde entier. Pour activer le support GIF dans Qt, transmettez l'option de ligne de commande `-qt-gif` au script `configure` ou définissez l'option appropriée dans le programme d'installation de Qt.

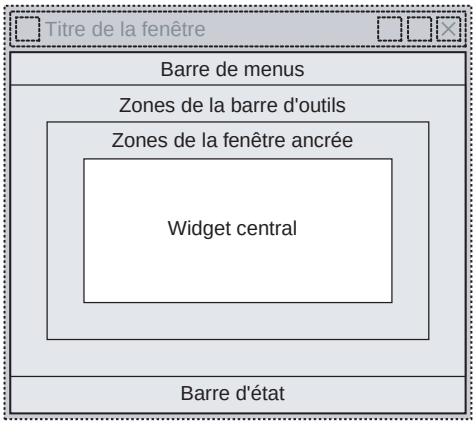


Figure 3.2
Les zones de QMainWindow

Dans notre cas, nous employons le mécanisme des ressources de Qt, puisqu'il s'avère plus pratique que de charger des fichiers à l'exécution et il est compatible avec n'importe quel format d'image pris en charge. Nous avons choisi de stocker les images dans l'arborescence source dans un sous-répertoire nommé `images`.

Pour utiliser le système des ressources de Qt, nous devons créer un fichier de ressources et ajouter une ligne au fichier `.pro` qui identifie le fichier de ressources. Dans cet exemple, nous avons appelé le fichier de ressources `spreadsheet.qrc`, nous insérons donc la ligne suivante dans le fichier `.pro` :

```
RESOURCES = spreadsheet.qrc
```

Le fichier de ressources lui-même utilise un format XML simple. Voici un extrait de celui que nous avons employé :

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/icon.png</file>
    ...
    <file>images/gotocell.png</file>
</qresource>
</RCC>
```

Les fichiers de ressources sont compilés dans l'exécutable de l'application, vous ne pouvez donc pas les perdre. Quand vous vous référez aux ressources, vous codez le préfixe :/ (double point, slash), c'est pourquoi l'icône est spécifiée comme suit, `:/images/icon.png`. Les ressources peuvent être de n'importe quel type (pas uniquement des images) et vous avez la possibilité de les utiliser à la plupart des emplacements où Qt attend un nom de fichier. Vous les étudierez plus en détail au Chapitre 12.

Créer des menus et des barres d'outils

La majorité des applications GUI modernes proposent des menus, des menus contextuels et des barres d'outils. Les menus permettent aux utilisateurs d'explorer l'application et d'apprendre à connaître de nouvelles commandes, alors que les menus contextuels et les barres d'outils fournissent un accès rapide aux fonctionnalités fréquemment utilisées.

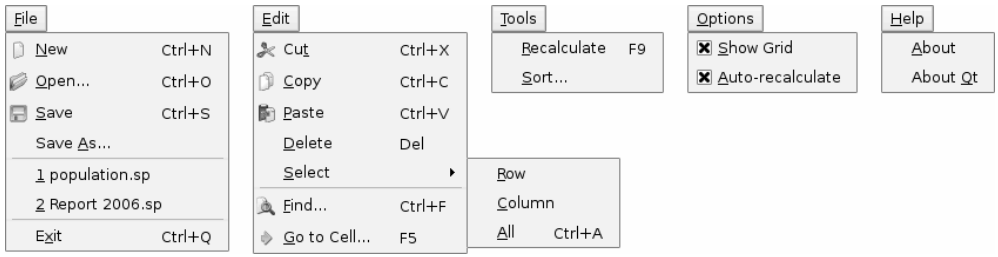


Figure 3.3
Les menus de l'application Spreadsheet

Qt simplifie la programmation des menus et des barres d'outils grâce à son concept d'action. Une *action* est un élément qui peut être ajouté à n'importe quel nombre de menus et barres d'outils. Créer des menus et des barres d'outils dans Qt implique ces étapes :

- créer et configurer les actions ;
- créer des menus et y introduire des actions ;
- créer des barres d'outils et y introduire des actions.

Dans l'application Spreadsheet, les actions sont créées dans `createActions() :`

```
void MainWindow::createActions()
{
    newAction = new QAction(tr("&New"), this);
    newAction->setIcon(QIcon(":/images/new.png"));
    newAction->setShortcut(tr("Ctrl+N"));
    newAction->setStatusTip(tr("Create a new spreadsheet file"));
    connect(newAction, SIGNAL(triggered()), this, SLOT(newFile()));
```

L'action `New` a un bouton d'accès rapide (`New`), un parent (la fenêtre principale), une icône (`new.png`), un raccourci clavier (Ctrl+N) et une infobulle liée à l'état. Nous connectons le signal `triggered()` de l'action au slot privé `newFile()` de la fenêtre principale, que nous implémenterons dans la prochaine section. Cette connexion garantit que lorsque l'utilisateur sélectionne File > New, clique sur le bouton `New` de la barre d'outils, ou appuie sur Ctrl+N, le slot `newFile()` est appelé.

Les actions `Open`, `Save` et `Save As` ressemblent beaucoup à l'action `New`, nous passerons donc directement à la partie "fichiers ouverts récemment" du menu File :

```
...
for (int i = 0; i < MaxRecentFiles; ++i) {
    recentFileActions[i] = new QAction(this);
    recentFileActions[i]->setVisible(false);
    connect(recentFileActions[i], SIGNAL(triggered()),
            this, SLOT(openRecentFile()));
}
```

Nous alimentons le tableau `recentFileActions` avec des actions. Chaque action est masquée et connectée au slot `openRecentFile()`. Plus tard, nous verrons comment afficher et utiliser les actions relatives aux fichiers récents.

Nous pouvons donc passer à l'action `Select All` :

```
...
selectAllAction = new QAction(tr("&All"), this);
selectAllAction->setShortcut(tr("Ctrl+A"));
selectAllAction->setStatusTip(tr("Select all the cells in the "
                                 "spreadsheet"));
connect(selectAllAction, SIGNAL(triggered()),
        spreadsheet, SLOT(selectAll()));
```

Le slot `selectAll()` est fourni par l'un des ancêtres de `QTableWidget`, `QAbstractItemView`, nous n'avons donc pas à l'implémenter nous-mêmes.

Continuons donc par l'action `Show Grid` dans le menu Options :

```
...
showGridAction = new QAction(tr("&Show Grid"), this);
showGridAction->setCheckable(true);
showGridAction->setChecked(spreadsheet->showGrid());
showGridAction->setStatusTip(tr("Show or hide the spreadsheet's "
                                "grid"));
connect(showGridAction, SIGNAL(toggled(bool)),
        spreadsheet, SLOT(setShowGrid(bool)));
```

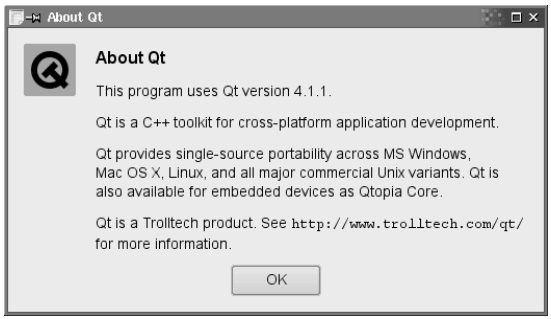
`Show Grid` est une action à cocher. Elle est affichée avec une coche dans le menu et est implémentée comme un bouton bascule dans la barre d'outils. Quand l'action est activée, le composant `Spreadsheet` affiche une grille. Nous initialisons l'action avec la valeur par défaut du composant `Spreadsheet`, de sorte qu'elles soient synchronisées au démarrage. Puis nous connectons le signal `toggled(bool)` de l'action `Show Grid` au slot `setShowGrid(bool)` du composant `Spreadsheet`, qu'il hérite de `QTableWidget`. Lorsque cette action est ajoutée à un menu ou à une barre d'outils, l'utilisateur peut activer ou désactiver l'affichage de la grille.

Les actions `Show Grid` et `Auto-Recalculate` sont des actions à cocher indépendantes. Qt prend aussi en charge des actions qui s'excluent mutuellement par le biais de la classe `QActionGroup`.

```
    ...
    aboutQtAction = new QAction(tr("About &Qt"), this);
    aboutQtAction->setStatusTip(tr("Show the Qt library's About box"));
    connect(aboutQtAction, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
}
```

Concernant l'action `About Qt`, nous utilisons le slot `aboutQt()` de l'objet `QApplication`, accessible *via* la variable globale `qApp`.

Figure 3.4
About Qt



Maintenant que nous avons créé les actions, nous pouvons poursuivre en concevant un système de menus qui les englobe :

```
void MainWindow::createMenus()
{
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(newAction);
    fileMenu->addAction(openAction);
    fileMenu->addAction(saveAction);
    fileMenu->addAction(saveAsAction);
    separatorAction = fileMenu->addSeparator();
    for (int i = 0; i < MaxRecentFiles; ++i)
        fileMenu->addAction(recentFileActions[i]);
    fileMenu->addSeparator();
    fileMenu->addAction(exitAction);
```

Dans Qt, les menus sont des instances de `QMenu`. La fonction `addMenu()` crée un widget `QMenu` avec le texte spécifié et l'ajoute à la barre de menus. La fonction `QMainWindow::menuBar()` retourne un pointeur vers un `QMenuBar`. La barre de menus est créée la première fois que `menuBar()` est appelé.

Nous commençons par créer le menu File, puis nous y ajoutons les actions `New`, `Open`, `Save` et `Save As`. Nous insérons un séparateur pour regrouper visuellement des éléments connexes. Nous utilisons une boucle `for` pour ajouter les actions (masquées à l'origine) du tableau `recentFileActions`, puis nous ajoutons l'action `exitAction` à la fin.

Nous avons conservé un pointeur vers l'un des séparateurs. Nous avons ainsi la possibilité de le masquer (s'il n'y a pas de fichiers récents) ou de l'afficher, parce que nous ne voulons pas afficher deux séparateurs sans rien entre eux.

```
    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(cutAction);
    editMenu->addAction(copyAction);
    editMenu->addAction(pasteAction);
    editMenu->addAction(deleteAction);

    selectSubMenu = editMenu->addMenu(tr("&Select"));
    selectSubMenu->addAction(selectRowAction);
    selectSubMenu->addAction(selectColumnAction);
    selectSubMenu->addAction(selectAllAction);

    editMenu->addSeparator();
    editMenu->addAction(findAction);
    editMenu->addAction(goToCellAction);
```

Occupons-nous désormais de créer le menu Edit, en ajoutant des actions avec `QMenu::addAction()` comme nous l'avons fait pour le menu File et en ajoutant le sous-menu avec `QMenu::addMenu()` à l'endroit où nous souhaitons qu'il apparaisse. Le sous-menu, comme le menu auquel il appartient, est un `QMenu`.

```
    toolsMenu = menuBar()->addMenu(tr("&Tools"));
    toolsMenu->addAction(recalculateAction);
    toolsMenu->addAction(sortAction);

    optionsMenu = menuBar()->addMenu(tr("&Options"));
    optionsMenu->addAction(showGridAction);
    optionsMenu->addAction(autoRecalcAction);

    menuBar()->addSeparator();

    helpMenu = menuBar()->addMenu(tr("&Help"));
    helpMenu->addAction(aboutAction);
    helpMenu->addAction(aboutQtAction);
}
```

Nous créons les menus Tools, Options et Help de manière similaire. Nous introduisons un séparateur entre les menus Options et Help. En styles Motif et CDE, le séparateur aligne le menu Help à droite ; dans les autres styles, le séparateur est ignoré.

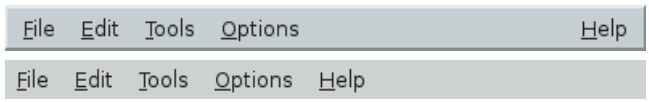


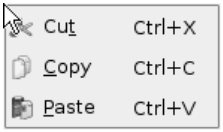
Figure 3.5
Barre de menus en styles Motif et Windows

```
void MainWindow::createContextMenu()
{
    spreadsheet->addAction(cutAction);
```

```
    spreadsheet->addAction(copyAction);
    spreadsheet->addAction(pasteAction);
    spreadsheet->setContextMenuPolicy(Qt::ActionsContextMenu);
}
```

Tout widget Qt peut avoir une liste de `QAction` associée. Pour proposer un menu contextuel pour l'application, nous ajoutons les actions souhaitées au widget `Spreadsheet` et nous définissons la stratégie du menu contextuel de ce widget de sorte qu'il affiche un menu contextuel avec ces actions. Les menus contextuels sont invoqués en cliquant du bouton droit sur un widget ou en appuyant sur une touche spécifique à la plate-forme.

Figure 3.6
Le menu contextuel de l'application Spreadsheet



Il existe un moyen plus élaboré de proposer des menus contextuels : implémenter à nouveau la fonction `QWidget::contextMenuEvent()`, créer un widget `QMenu`, l'alimenter avec les actions voulues et appeler `exec()`.

```
void MainWindow::createToolBars()
{
    fileToolBar = addToolBar(tr("&File"));
    fileToolBar->addAction(newAction);
    fileToolBar->addAction(openAction);
    fileToolBar->addAction(saveAction);

    editToolBar = addToolBar(tr("&Edit"));
    editToolBar->addAction(cutAction);
    editToolBar->addAction(copyAction);
    editToolBar->addAction(pasteAction);
    editToolBar->addSeparator();
    editToolBar->addAction(findAction);
    editToolBar->addAction(goToCellAction);
}
```

La création de barres d'outils ressemble beaucoup à celle des menus. Nous concevons les barres d'outils File et Edit. Comme un menu, une barre d'outils peut posséder des séparateurs.

Figure 3.7
Les barres d'outils de l'application Spreadsheet

Configurer la barre d'état

Lorsque les menus et les barres d'outils sont terminés, vous êtes prêt à vous charger de la barre d'état de l'application Spreadsheet.

Normalement, cette barre d'état contient deux indicateurs : l'emplacement et la formule de la cellule en cours.

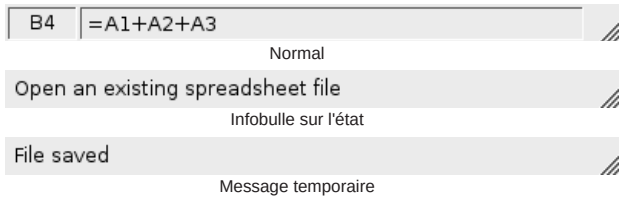


Figure 3.8
La barre d'état de l'application Spreadsheet

Le constructeur de `MainWindow` appelle `createStatusBar()` pour configurer la barre d'état :

```
void MainWindow::createStatusBar()
{
    locationLabel = new QLabel(" W999 ");
    locationLabel->setAlignment(Qt::AlignHCenter);
    locationLabel->setMinimumSize(locationLabel->sizeHint());

    formulaLabel = new QLabel;
    formulaLabel->setIndent(3);

    statusBar()->addWidget(locationLabel);
    statusBar()->addWidget(formulaLabel, 1);

    connect(spreadsheet, SIGNAL(currentCellChanged(int, int, int, int)),
            this, SLOT(updateStatusBar()));
    connect(spreadsheet, SIGNAL(modified()),
            this, SLOT(spreadsheetModified()));

    updateStatusBar();
}
```

La fonction `QMainWindow::statusBar()` retourne un pointeur vers la barre d'état. (La barre d'état est créée la première fois que `statusBar()` est appelée.) Les indicateurs d'état sont simplement des `QLabel` dont nous modifions le texte dès que cela s'avère nécessaire. Nous avons ajouté une indentation à `formulaLabel`, pour que le texte qui y est affiché soit légèrement décalé du bord gauche. Quand les `QLabel` sont ajoutés à la barre d'état, ils sont automatiquement reparentés pour devenir des enfants de cette dernière.

La Figure 3.8 montre que les deux étiquettes ont des exigences différentes s'agissant de l'espace. L'indicateur relatif à l'emplacement de la cellule ne nécessite que très peu de place, et lorsque la fenêtre est redimensionnée, tout espace supplémentaire devrait revenir à l'indicateur de la formule de la cellule sur la droite. Vous y parvenez en spécifiant un facteur de redimensionnement

de 1 dans l'appel `QStatusBar::addWidget()` de l'étiquette de la formule. L'indicateur d'emplacement présente un facteur de redimensionnement par défaut de 0, ce qui signifie qu'il préfère ne pas être étiré.

Quand `QStatusBar` organise l'affichage des widgets indicateur, il essaie de respecter la taille idéale de chaque widget spécifiée par `QWidget::sizeHint()`, puis redimensionne tout widget étirable pour combler l'espace disponible. La taille idéale d'un widget dépend du contenu de ce widget et varie en fonction des modifications du contenu. Pour éviter de redimensionner constamment l'indicateur d'emplacement, nous configurons sa taille minimale de sorte qu'elle suffise pour contenir le texte le plus grand possible ("W999"), avec très peu d'espace supplémentaire. Nous définissons aussi son alignement en `Qt::AlignHCenter` pour centrer le texte horizontalement.

Vers la fin de la fonction, nous connectons deux des signaux de `Spreadsheet` à deux des slots de `MainWindow` : `updateStatusBar()` et `spreadsheetModified()`.

```
void MainWindow::updateStatusBar()
{
    locationLabel->setText(spreadsheet->currentLocation());
    formulaLabel->setText(spreadsheet->currentFormula());
}
```

Le slot `updateStatusBar()` met à jour les indicateurs relatifs à l'emplacement et à la formule de la cellule. Il est invoqué dès que l'utilisateur déplace le curseur vers une autre cellule. Le slot sert également de fonction ordinaire à la fin de `createStatusBar()` pour initialiser les indicateurs. Il se révèle nécessaire puisque Spreadsheet n'émet pas le signal `currentCellChanged()` au démarrage.

```
void MainWindow::spreadsheetModified()
{
    setWindowModified(true);
    updateStatusBar();
}
```

Le slot `spreadsheetModified()` définit la propriété `windowModified` en `true`, ce qui met à jour la barre de titre. La fonction met également à jour les indicateurs d'emplacement et de formule, pour qu'ils reflètent les circonstances actuelles.

Implémenter le menu File

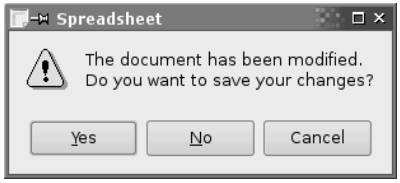
Dans cette section, nous allons implémenter les slots et les fonctions privées nécessaires pour faire fonctionner les options du menu File et pour gérer la liste des fichiers ouverts récemment.

```
void MainWindow::newFile()
{
    if (okToContinue()) {
        spreadsheet->clear();
```

```
            setCurrentFile("");
        }
    }
```

Le slot `newFile()` est appelé lorsque l'utilisateur clique sur l'option File > New ou sur le bouton `New` de la barre d'outils. La fonction privée `okToContinue()` demande à l'utilisateur s'il désire enregistrer ses modifications, si certaines modifications n'ont pas été sauvegardées (voir Figure 3.9). Elle retourne `true` si l'utilisateur choisit Yes ou No (vous enregistrez le document en appuyant sur Yes), et `false` si l'utilisateur clique sur Cancel. La fonction `Spreadsheet::clear()` efface toutes les cellules et formules du tableur. La fonction privée `setCurrentFile()` met à jour le titre de la fenêtre pour indiquer qu'un document sans titre est en train d'être modifié, en plus de configurer la variable privée `curFile` et de mettre à jour la liste des fichiers ouverts récemment.

Figure 3.9
"Voulez-vous enregistrer vos modifications ?"




```
bool MainWindow::okToContinue()
{
   if (isWindowModified()) {
      int r = QMessageBox::warning(this, tr("Spreadsheet"),
                    tr("The document has been modified.\n"
                       "Do you want to save your changes?"),
                    QMessageBox::Yes | QMessageBox::Default,
                    QMessageBox::No,
                    QMessageBox::Cancel | QMessageBox::Escape);
      if (r == QMessageBox::Yes) {
         return save();
      } else if (r == QMessageBox::Cancel) {
         return false;
      }
   }
   return true;
}
```

Dans `okToContinue()`, nous contrôlons l'état de la propriété `windowModified`. S'il est correct, nous affichons la boîte de message illustrée en Figure 3.9. Celle-ci propose les boutons Yes, No et Cancel. `QMessageBox::Default` définit le bouton Yes comme bouton par défaut. `QMessageBox::Escape` définit la touche Echap comme synonyme de Cancel.

L'appel de `warning()` peut sembler assez complexe de prime abord, mais la syntaxe générale est simple :

```
QMessageBox::warning(parent, titre, message, bouton0, bouton1, ...);
```

`QMessageBox` propose aussi `information()`, `question()` et `critical()`, chacun possédant sa propre icône.

Figure 3.10
Icônes de boîte de message

```
void MainWindow::open()
{
   if (okToContinue()) {
      QString fileName = QFileDialog::getOpenFileName(this,
                                 tr("Open Spreadsheet"), ".",
                                 tr("Spreadsheet files (*.sp)"));
      if (!fileName.isEmpty())
         loadFile(fileName);
   }
}
```

Le slot `open()` correspond à File > Open. Comme `newFile()`, il appelle d'abord `okToContinue()` pour gérer toute modification non sauvegardée. Puis il utilise la fonction statique `QFileDialog::getOpenFileName()` très pratique pour demander le nom du nouveau fichier à l'utilisateur. La fonction ouvre une boîte de dialogue, permet à l'utilisateur de choisir un fichier et retourne le nom de ce dernier – ou une chaîne vide si l'utilisateur clique sur Cancel.

Le premier argument de `QFileDialog::getOpenFileName()` est le widget parent. La relation parent-enfant ne signifie pas la même chose pour les boîtes de dialogue et pour les autres widgets. Une boîte de dialogue est toujours une fenêtre en soi, mais si elle a un parent, elle est centrée en haut de ce dernier par défaut. Une boîte de dialogue enfant partage aussi l'entrée de la barre des tâches de son parent.

Le second argument est le titre que la boîte de dialogue doit utiliser. Le troisième argument indique le répertoire depuis lequel il doit démarrer, dans notre cas le répertoire en cours.

Le quatrième argument spécifie les filtres de fichier. Un filtre de fichier est constitué d'un texte descriptif et d'un modèle générique. Si nous avions pris en charge les fichiers de valeurs séparées par des virgules et les fichiers Lotus 1-2-3, en plus du format de fichier natif de Spreadsheet, nous aurions employé le filtre suivant :

```
tr("Spreadsheet files (*.sp)\n"
   "Comma-separated values files (*.csv)\n"
   "Lotus 1-2-3 files (*.wk1 *.wks)")
```

La fonction privée `loadFile()` a été invoquée dans `open()` pour charger le fichier. Nous en avons fait une fonction indépendante, parce que nous aurons besoin de la même fonctionnalité pour charger les fichiers ouverts récemment :

```
bool MainWindow::loadFile(const QString &fileName)
{
   if (!spreadsheet->readFile(fileName)) {
```

```
        statusBar()->showMessage(tr("Loading canceled"), 2000);
        return false;
    }

    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File loaded"), 2000);
    return true;
}
```

Nous utilisons `Spreadsheet::readFile()` pour lire le fichier sur le disque. Si le chargement est effectué avec succès, nous appelons `setCurrentFile()` pour mettre à jour le titre de la fenêtre ; sinon, `Spreadsheet::readFile()` aurait déjà informé l'utilisateur du problème par une boîte de message. En général, il est recommandé de laisser les composants de bas niveau émettre des messages d'erreur, parce qu'ils peuvent apporter des détails précis sur ce qui s'est passé.

Dans les deux cas, nous affichons un message dans la barre d'état pendant 2 secondes (2000 millisecondes) pour informer l'utilisateur des tâches effectuées par l'application.

```
bool MainWindow::save()
{
    if (curFile.isEmpty()) {
        return saveAs();
    } else {
        return saveFile(curFile);
    }
}

bool MainWindow::saveFile(const QString &fileName)
{
    if (!spreadsheet->writeFile(fileName)) {
        statusBar()->showMessage(tr("Saving canceled"), 2000);
        return false;
    }

    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File saved"), 2000);
    return true;
}
```

Le slot `save()` correspond à File > Save. Si le fichier porte déjà un nom puisque il a déjà été ouvert ou enregistré, `save()` appelle `saveFile()` avec ce nom ; sinon il invoque simplement `saveAs()`.

```
bool MainWindow::saveAs()
{
    QString fileName = QFileDialog::getSaveFileName(this,
                                tr("Save Spreadsheet"), ".",
                                tr("Spreadsheet files (*.sp)"));
    if (fileName.isEmpty())
        return false;
```

```
    return saveFile(fileName);
}
```

Le slot `saveAs()` correspond à File > Save As. Nous appelons `QFileDialog::getSaveFileName()` pour que l'utilisateur indique un nom de fichier. Si l'utilisateur clique sur Cancel, nous retournons `false`, qui est ensuite transmis à son appelant (`save()` ou `okToContinue()`).

Si le fichier existe déjà, la fonction `getSaveFileName()` demandera à l'utilisateur de confirmer qu'il veut bien le remplacer. Ce comportement peut être modifié en transmettant `QFileDialog::DontConfirmOverwrite` comme argument supplémentaire à `getSaveFileName()`.

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    if (okToContinue()) {
        writeSettings();
        event->accept();
    } else {
        event->ignore();
    }
}
```

Quand l'utilisateur clique sur File > Exit ou sur le bouton de fermeture dans la barre de titre de la fenêtre, le slot `QWidget::close()` est invoqué. Un événement "close" est donc envoyé au widget. En implémentant à nouveau `QWidget::closeEvent()`, nous avons la possibilité de fermer la fenêtre principale et de décider si nous voulons aussi fermer la fenêtre ou non.

Si certaines modifications n'ont pas été enregistrées et si l'utilisateur sélectionne Cancel, nous "ignorons" l'événement et la fenêtre n'en sera pas affectée. En temps normal, nous acceptons l'événement ; Qt masque donc la fenêtre. Nous invoquons également la fonction privée `writeSettings()` afin de sauvegarder les paramètres en cours de l'application.

Quand la dernière fenêtre est fermée, l'application se termine. Si nécessaire, nous pouvons désactiver ce comportement en configurant la propriété `quitOnLastWindowClosed` de `QApplication` en `false`, auquel cas l'application continue à être exécutée jusqu'à ce que nous appelions `QApplication::quit()`.

```
void MainWindow::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    setWindowModified(false);

    QString shownName = "Untitled";

    if (!curFile.isEmpty()) {
        shownName = strippedName(curFile);
        recentFiles.removeAll(curFile);
        recentFiles.prepend(curFile);
        updateRecentFileActions();
    }
```

```
    setWindowTitle(tr("%1[*] - %2").arg(shownName)
                                   .arg(tr("Spreadsheet")));
}

QString MainWindow::strippedName(const QString &fullFileName)
{
    return QFileInfo(fullFileName).fileName();
}
```

Dans `setCurrentFile()`, nous définissons la variable privée `curFile` qui stocke le nom du fichier en cours de modification. Avant d'afficher le nom du fichier dans la barre de titre, nous supprimons le chemin d'accès du fichier avec `strippedName()` pour le rendre plus convivial.

Chaque `QWidget` possède une propriété `windowModified` qui doit être définie en `true` si le document présente des modifications non sauvegardées et en `false` dans les autres cas. Sous Mac OS X, les documents non sauvegardés sont indiqués par un point dans le bouton de fermeture de la barre de titre de la fenêtre ; sur les autres plates-formes, ils sont indiqués par un astérisque après le nom de fichier. Qt se charge automatiquement de ce comportement, tant que nous mettons à jour la propriété `windowModified` et que nous plaçons le marqueur "[*]" dans le titre de la fenêtre à l'endroit où nous souhaitons voir apparaître l'astérisque.

Le texte transmis à la fonction `setWindowTitle()` était le suivant :

```
tr("%1[*] - %2").arg(shownName)
                .arg(tr("Spreadsheet"))
```

La fonction `QString::arg()` remplace le paramètre "%*n*" de numéro le plus bas par son argument et retourne la chaîne ainsi obtenue. Dans ce cas, `arg()` est utilisé avec deux paramètres "%*n*". Le premier appel de `arg()` remplace "%1" ; le second appel remplace "%2".Si le nom de fichier est "budget.sp" et qu'aucun fichier de traduction n'est chargé, la chaîne obtenue serait "budget.sp[*] – Spreadsheet".Il aurait été plus simple d'écrire

```
setWindowTitle(shownName + tr("[*] - Spreadsheet"));
```

mais `arg()` offre une plus grande souplesse pour les traducteurs.

S'il existe un nom de fichier, nous mettons à jour `recentFiles`, la liste des fichiers ouverts récemment. Nous invoquons `removeAll()` pour supprimer toutes les occurrences du nom de fichier dans la liste afin d'éviter les copies ; puis nous appelons `prepend()` pour ajouter le nom de fichier en tant que premier élément. Après la mise à jour de la liste, nous appelons la fonction privée `updateRecentFileActions()` de manière à mettre à jour les entrées dans le menu File.

```
void MainWindow::updateRecentFileActions()
{
    QMutableStringListIterator i(recentFiles);
    while (i.hasNext()) {
        if (!QFile::exists(i.next()))
            i.remove();
```

```
    }

    for (int j = 0; j < MaxRecentFiles; ++j) {
       if (j < recentFiles.count()) {
          QString text = tr("&%1 %2")
                         .arg(j + 1)
                         .arg(strippedName(recentFiles[j]));
          recentFileActions[j]->setText(text);
          recentFileActions[j]->setData(recentFiles[j]);
          recentFileActions[j]->setVisible(true);
       } else {
          recentFileActions[j]->setVisible(false);
       }
    }
    separatorAction->setVisible(!recentFiles.isEmpty());
}
```

Nous commençons par supprimer tout fichier qui n'existe plus à l'aide d'un itérateur de style Java. Certains fichiers peuvent avoir été utilisés dans une session antérieure, mais ont été supprimés depuis. La variable `recentFiles` est de type `QStringList` (liste de `QString`). Le Chapitre 11 étudie en détail les classes conteneur comme `QStringList` vous expliquant comment elles sont liées à la bibliothèque C++ STL (Standard Template Library), et vous explique comment employer les classes d'itérateurs de style Java dans Qt.

Nous parcourons ensuite à nouveau la liste des fichiers, mais cette fois-ci en utilisant une indexation de style tableau. Pour chaque fichier, nous créons une chaîne composée d'un caractère &, d'un chiffre (`j + 1`), d'un espace et du nom de fichier (sans son chemin d'accès). Nous définissons l'action correspondante pour qu'elle utilise ce texte. Par exemple, si le premier fichier était C:\My Documents\tab04.sp, le texte de la première action serait "&1 tab04.sp".

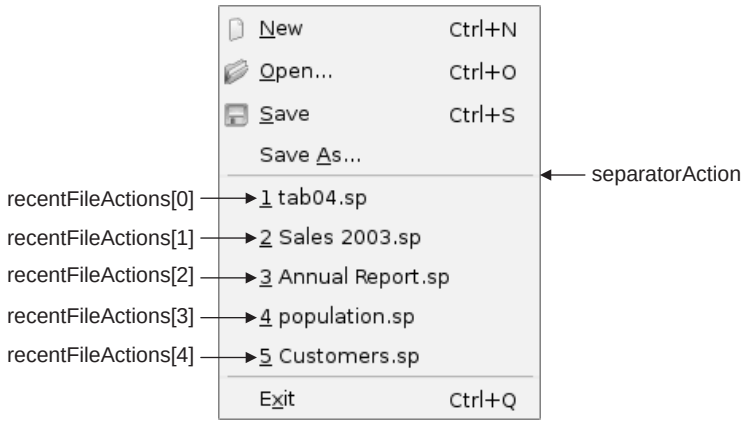


Figure 3.11
Le menu File avec les fichiers ouverts récemment

Chaque action peut avoir un élément "donnée" associé de type `QVariant`. Le type `QVariant` peut contenir des valeurs de plusieurs types C++ et Qt ; c'est expliqué au Chapitre 11. Ici, nous

stockons le nom complet du fichier dans l'élément "donnée" de l'action, pour pouvoir le récupérer facilement par la suite. Nous configurons également l'action de sorte qu'elle soit visible.

S'il y a plus d'actions de fichiers que de fichiers récents, nous masquons simplement les actions supplémentaires. Enfin, s'il y a au moins un fichier récent, nous définissons le séparateur pour qu'il s'affiche.

```
void MainWindow::openRecentFile()
{
   if (okToContinue()) {
      QAction *action = qobject_cast<QAction *>(sender());
      if (action)
         loadFile(action->data().toString());
   }
}
```

Quand l'utilisateur sélectionne un fichier récent, le slot `openRecentFile()` est appelé. La fonction `okToContinue()` est exécutée si des changements n'ont pas été sauvegardés, et si l'utilisateur n'annule pas, nous identifions quelle action a appelé le slot grâce à `QObject::sender()`.

La fonction `qobject_cast<T>()` accomplit une conversion dynamique basée sur les méta-informations générées par `moc`, le compilateur des méta-objets de Qt. Elle retourne un pointeur vers la sous-classe `QObject` demandée, ou 0 si l'objet n'a pas pu être converti dans ce type. Contrairement à `dynamic_cast<T>()` du langage C++ standard, `qobject_cast<T>()` de Qt fonctionne correctement dans les bibliothèques dynamiques. Dans notre exemple, nous utilisons `qobject_cast<T>()` pour convertir un pointeur `QObject` en une action `QAction`. Si la conversion a été effectuée avec succès (ce devrait être le cas), nous appelons `loadFile()` avec le nom complet du fichier que nous extrayons des données de l'action.

Notez qu'étant donné que nous savons que l'expéditeur est de type `QAction`, le programme fonctionnerait toujours si nous avions utilisé `static_cast<T>()` ou une conversion traditionnelle de style C. Consultez la section "Conversions de type" en Annexe B pour connaître les diverses conversions C++.

Utiliser des boîtes de dialogue

Dans cette section, nous allons vous expliquer comment utiliser des boîtes de dialogue dans Qt – comment les créer et les initialiser, les exécuter et répondre aux sélections effectuées par l'utilisateur interagissant avec elles. Nous emploierons les boîtes de dialogue Find, Go-to-Cell et Sort créées au Chapitre 2. Nous créerons aussi une boîte simple About.

Nous commençons par la boîte de dialogue Find. Nous voulons que l'utilisateur puisse basculer à volonté entre la fenêtre principale Spreadsheet et la boîte de dialogue Find, cette dernière doit donc être non modale. Une fenêtre *non modale* est exécutée indépendamment de toute autre fenêtre dans l'application.

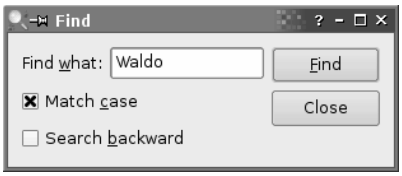


Figure 3.12
La boîte de dialogue Find de l'application Spreadsheet

Lorsque des boîtes de dialogue non modales sont créées, leurs signaux sont normalement connectés aux slots qui répondent aux interactions de l'utilisateur.

```
void MainWindow::find()
{
   if (!findDialog) {
      findDialog = new FindDialog(this);
      connect(findDialog, SIGNAL(findNext(const QString &,
                                          Qt::CaseSensitivity)),
              spreadsheet, SLOT(findNext(const QString &,
                                         Qt::CaseSensitivity)));
      connect(findDialog, SIGNAL(findPrevious(const QString &,
                                              Qt::CaseSensitivity)),
              spreadsheet, SLOT(findPrevious(const QString &,
                                             Qt::CaseSensitivity)));
   }

   findDialog->show();
   findDialog->activateWindow();
}
```

La boîte de dialogue Find est une fenêtre qui permet à l'utilisateur de rechercher du texte dans le tableur. Le slot `find()` est appelé lorsque l'utilisateur clique sur Edit > Find pour ouvrir la boîte de dialogue Find. A ce stade, plusieurs scénarios sont possibles :

- C'est la première fois que l'utilisateur appelle la boîte de dialogue Find.
- La boîte de dialogue Find a déjà été appelée auparavant, mais l'utilisateur l'a fermée.
- La boîte de dialogue Find a déjà été appelée auparavant et est toujours affichée.

Si la boîte de dialogue Find n'existe pas encore, nous la créons et nous connectons ses signaux `findNext()` et `findPrevious()` aux slots `Spreadsheet` correspondants. Nous aurions aussi pu créer la boîte de dialogue dans le constructeur de `MainWindow`, mais ajourner sa création rend le démarrage plus rapide. De même, si la boîte de dialogue n'est jamais utilisée, elle n'est jamais créée, ce qui vous fait gagner du temps et de la mémoire.

Nous invoquons ensuite `show()` et `activateWindow()` pour nous assurer que la fenêtre est visible et active. Un seul appel de `show()` est suffisant pour afficher et activer une fenêtre masquée, mais la boîte de dialogue Find peut être appelée quand sa fenêtre est déjà visible, auquel cas `show()` ne fait rien et `activateWindow()` est nécessaire pour activer la fenêtre. Vous auriez aussi pu écrire

```
if (findDialog->isHidden()) {
```

```
        findDialog->show();
    } else {
        findDialog->activateWindow();
    }
```

Ce code revient à regarder des deux côtés d'une route à sens unique avant de traverser.

Nous allons à présent analyser la boîte de dialogue Go-to-Cell. Nous voulons que l'utilisateur l'ouvre, l'utilise, puis la ferme sans pouvoir basculer vers d'autres fenêtres dans l'application. Cela signifie que la boîte de dialogue Go-to-Cell doit être modale. Une fenêtre *modale* est une fenêtre qui s'affiche quand elle est appelée et bloque l'application. Tout autre traitement ou interaction est impossible tant que la fenêtre n'est pas fermée. Les boîtes de dialogue d'ouverture de fichier et les boîtes de message utilisées précédemment étaient modales.

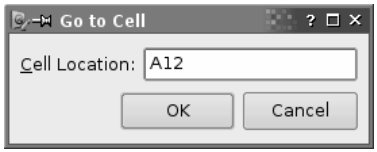


Figure 3.13
La boîte de dialogue Go-to-Cell de l'application Spreadsheet

Une boîte de dialogue n'est pas modale si elle est appelée à l'aide de `show()` (à moins que nous appelions `setModal()` au préalable pour la rendre modale) ; elle est modale si elle est invoquée avec `exec()`.

```
void MainWindow::goToCell()
{
    GoToCellDialog dialog(this);
    if (dialog.exec()) {
        QString str = dialog.lineEdit->text().toUpper();
        spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
                                    str[0].unicode() - 'A');
    }
}
```

La fonction `QDialog::exec()` retourne une valeur `true` (`QDialog::Accepted`) si la boîte de dialogue est acceptée, et une valeur `false` (`QDialog::Rejected`) dans les autres cas. Souvenez-vous que lorsque nous avons créé la boîte de dialogue Go-to-Cell avec le *Qt Designer* au Chapitre 2, nous avons connecté OK à `accept()` et Cancel à `reject()`. Si l'utilisateur clique sur OK, nous définissons la cellule actuelle avec la valeur présente dans l'éditeur de lignes.

La fonction `QTableWidget::setCurrentCell()` reçoit deux arguments : un index des lignes et un index des colonnes. Dans l'application Spreadsheet, la cellule A1 correspond à la cellule (0, 0) et la cellule B27 à la cellule (26, 1). Pour obtenir l'index des lignes du `QString` retourné par `QLineEdit::text()`, nous devons extraire le nombre de lignes avec `QString::mid()` (qui retourne une sous-chaîne allant du début à la fin de la chaîne), la convertir en type `int` avec `QString::toInt()` et soustraire 1. Pour le nombre de colonnes, nous soustrayons la valeur numérique de 'A' de la valeur numérique du premier caractère en

majuscule de la chaîne. Nous savons que la chaîne aura le bon format parce que `QRegExpValidator` créé pour la boîte de dialogue n'autorise l'activation du bouton OK que s'il y a une lettre suivie par 3 chiffres maximum.

La fonction `goToCell()` diffère de tout le code étudié jusqu'à présent, puisqu'elle crée un widget (`GoToCellDialog`) sous la forme d'une variable sur la pile. En ajoutant une ligne, nous aurions pu utiliser tout aussi facilement `new` et `delete` :

```
void MainWindow::goToCell()
{
   GoToCellDialog *dialog = new GoToCellDialog(this);
   if (dialog->exec()) {
      QString str = dialog->lineEdit->text().toUpper();
      spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
                                  str[0].unicode() - 'A');
   }
   delete dialog;
}
```

La création de boîtes de dialogue modales (et de menus contextuels dans des réimplémentations `QWidget::contextMenuEvent()`) sur la pile est un modèle de programmation courant, parce qu'en règle générale, nous n'avons plus besoin de la boîte de dialogue (ou du menu) après l'avoir utilisée, et elle sera automatiquement détruite à la fin de la portée dans laquelle elle évolue.

Examinons maintenant la boîte de dialogue Sort. Celle-ci est une boîte de dialogue modale qui permet à l'utilisateur de trier la zone sélectionnée sur les colonnes qu'il spécifie. La Figure 3.14 montre un exemple de tri, avec la colonne B comme clé de tri primaire et la colonne A comme clé de tri secondaire (toutes les deux par ordre croissant).

Figure 3.14
Trier la zone sélectionnée du tableur

	A	B	C
1	George	Washington	1789-1797
2	John	Adams	1797-1801
3	Thomas	Jefferson	1801-1809
4	James	Madison	1809-1817
5	James	Monroe	1817-1825
6	John Quincy	Adams	1825-1829
7	Andrew	Jackson	1829-1837
8			

(b) Avant le tri

	A	B	C
1	John	Adams	1797-1801
2	John Quincy	Adams	1825-1829
3	Andrew	Jackson	1829-1837
4	Thomas	Jefferson	1801-1809
5	James	Madison	1809-1817
6	James	Monroe	1817-1825
7	George	Washington	1789-1797
8			

(b) Après le tri

```
void MainWindow::sort()
{
   SortDialog dialog(this);
   QTableWidgetSelectionRange range = spreadsheet->selectedRange();
   dialog.setColumnRange('A' + range.leftColumn(),
                         'A' + range.rightColumn());

   if (dialog.exec()) {
```

```
            SpreadsheetCompare compare;
            compare.keys[0] =
                  dialog.primaryColumnCombo->currentIndex();
            compare.keys[1] =
                  dialog.secondaryColumnCombo->currentIndex() - 1;
            compare.keys[2] =
                  dialog.tertiaryColumnCombo->currentIndex() - 1;
            compare.ascending[0] =
                  (dialog.primaryOrderCombo->currentIndex() == 0);
            compare.ascending[1] =
                  (dialog.secondaryOrderCombo->currentIndex() == 0);
            compare.ascending[2] =
                  (dialog.tertiaryOrderCombo->currentIndex() == 0);
            spreadsheet->sort(compare);
        }
    }
```

Le code dans `sort()` suit un modèle similaire à celui utilisé pour `goToCell()` :

- Nous créons la boîte de dialogue sur la pile et nous l'initialisons.
- Nous ouvrons la boîte de dialogue avec `exec()`.
- Si l'utilisateur clique sur OK, nous extrayons les valeurs saisies par ce dernier à partir des widgets de la boîte de dialogue et nous les utilisons.

L'appel de `setColumnRange()` définit les colonnes disponibles pour le tri sur les colonnes sélectionnées. Par exemple, en utilisant la sélection illustrée en Figure 3.14, `range.leftColumn()` produirait 0, ce qui fait 'A' + 0 = 'A', et `range.rightColumn()` produirait 2, ce qui fait 'A' + 2 = 'C'.

L'objet `compare` stocke les clés de tri primaire, secondaire et tertiaire, ainsi que leurs ordres de tri. (Nous verrons la définition de la classe `SpreadsheetCompare` dans le prochain chapitre.) L'objet est employé par `Spreadsheet::sort()` pour comparer deux lignes. Le tableau `keys` stocke les numéros de colonne des clés. Par exemple, si la sélection s'étend de C2 à E5, la colonne C correspond à la position 0. Le tableau `ascending` conserve l'ordre associé à chaque clé comme une valeur `bool`. `QComboBox::currentIndex()` retourne l'index de l'élément sélectionné, en commençant à 0. Concernant les clés secondaire et tertiaire, nous soustrayons un de l'élément en cours pour prendre en compte l'élément "None (Aucun)".

La fonction `sort()` répond à la demande, mais elle manque de fiabilité. Elle suppose que la boîte de dialogue Sort est implémentée de manière particulière, avec des zones de liste déroulante et des éléments "None". Cela signifie que si nous concevons à nouveau la boîte de dialogue Sort, nous devrions également réécrire ce code. Alors que cette approche convient pour une boîte de dialogue qui est toujours appelée depuis le même emplacement, elle conduit à un véritable cauchemar pour la maintenance si elle est employée à plusieurs endroits.

Une méthode plus fiable consiste à rendre la classe `SortDialog` plus intelligente en la faisant créer un objet `SpreadsheetCompare` auquel son appelant peut ensuite accéder. Cela simplifie significativement `MainWindow::sort()` :

```
void MainWindow::sort()
{
   SortDialog dialog(this);
   QTableWidgetSelectionRange range = spreadsheet->selectedRange();
   dialog.setColumnRange('A' + range.leftColumn(),
                         'A' + range.rightColumn());

   if (dialog.exec())
       spreadsheet->performSort(dialog.comparisonObject());
}
```

Cette approche conduit à des composants relativement indépendants et constitue presque toujours le bon choix pour des boîtes de dialogue qui seront invoquées depuis plusieurs emplacements.

Une technique plus radicale serait de transmettre un pointeur à l'objet `Spreadsheet` au moment de l'initialisation de l'objet `SortDialog` et de permettre à la boîte de dialogue d'opérer directement sur `Spreadsheet`. `SortDialog` devient donc moins général, parce qu'il ne fonctionnera que dans certains types de widgets, mais cela simplifie davantage le code en éliminant la fonction `SortDialog::setColumnRange()`. La fonction `MainWindow::sort()` devient donc

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    dialog.setSpreadsheet(spreadsheet);
    dialog.exec();
}
```

Cette approche reproduit la première : l'appelant n'a pas besoin de connaître la boîte de dialogue dans les moindres détails, mais c'est la boîte de dialogue qui doit totalement connaître les structures de données fournies par l'appelant. Cette technique peut être pratique quand la boîte de dialogue doit appliquer des changements en direct. Mais comme le code d'appel peu fiable de la première approche, cette troisième méthode ne fonctionne plus si les structures de données changent.

Certains développeurs choisissent une approche quant à l'utilisation des boîtes de dialogue et n'en changent plus. Cela présente l'avantage de favoriser la familiarité et la simplicité, parce que toutes leurs boîtes de dialogue respectent le même schéma, mais ils passent à côté des bénéfices apportés par les autres approches. La meilleure approche consiste à choisir la méthode au cas par cas.

Nous allons clore cette section avec la boîte de dialogue About. Nous pourrions créer une boîte de dialogue personnalisée comme pour les boîtes de dialogue Find ou Go-to-Cell pour présenter les informations relatives à l'application, mais vu que la plupart des boîtes About adoptent le même style, Qt propose une solution plus simple.

```
void MainWindow::about()
{
   QMessageBox::about(this, tr("About Spreadsheet"),
            tr("<h2>Spreadsheet 1.1</h2>"
```

```
                "<p>Copyright &copy; 2006 Software Inc."
                "<p>Spreadsheet is a small application that "
                "demonstrates QAction, QMainWindow, QMenuBar, "
                "QStatusBar, QTableWidget, QToolBar, and many other "
                "Qt classes."));
}
```

Vous obtenez la boîte About en appelant tout simplement la fonction statique `QMessageBox::about()`. Cette fonction ressemble beaucoup à `QMessageBox::warning()`, sauf qu'elle emploie l'icône de la fenêtre parent au lieu de l'icône standard d'avertissement.

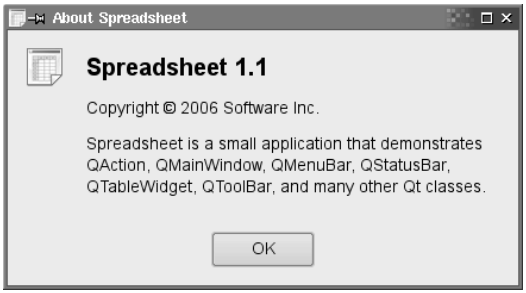


Figure 3.15
La boîte About de Spreadsheet

Jusqu'à présent, nous avons utilisé plusieurs fonctions statiques commodes dans `QMessageBox` et `QFileDialog`. Ces fonctions créent une boîte de dialogue, l'initialisent et appellent `exec()`. Il est également possible, même si c'est moins pratique, de créer un widget `QMessageBox` ou `QFileDialog` comme n'importe quel autre widget et d'appeler explicitement `exec()` ou même `show()`.

Stocker des paramètres

Dans le constructeur `MainWindow`, nous avons invoqué `readSettings()` afin de charger les paramètres stockés de l'application. De même, dans `closeEvent()`, nous avons appelé `writeSettings()` pour sauvegarder les paramètres. Ces deux fonctions sont les dernières fonctions membres `MainWindow` qui doivent être implémentées.

```
void MainWindow::writeSettings()
{
   QSettings settings("Software Inc.", "Spreadsheet");

   settings.setValue("geometry", geometry());
   settings.setValue("recentFiles", recentFiles);
   settings.setValue("showGrid", showGridAction->isChecked());
   settings.setValue("autoRecalc", autoRecalcAction->isChecked());
}
```

La fonction `writeSettings()` enregistre la disposition (position et taille) de la fenêtre principale, la liste des fichiers ouverts récemment et les options `Show Grid` et `Auto-Recalculate`.

Par défaut, `QSettings` stocke les paramètres de l'application à des emplacements spécifiques à la plate-forme. Sous Windows, il utilise le registre du système ; sous Unix, il stocke les données dans des fichiers texte ; sous Mac OS X, il emploie l'API des préférences de Core Foundation.

Les arguments du constructeur spécifient les noms de l'organisation et de l'application. Ces informations sont exploitées d'une façon spécifique à la plate-forme pour trouver un emplacement aux paramètres.

`QSettings` stocke les paramètres sous forme de paires *clé-valeur*. La *clé* est similaire au chemin d'accès du système de fichiers. Des sous-clés peuvent être spécifiées grâce à une syntaxe de style chemin d'accès (par exemple, findDialog/matchCase) ou à `beginGroup()` et `endGroup()` :

```
settings.beginGroup("findDialog");
settings.setValue("matchCase", caseCheckBox->isChecked());
settings.setValue("searchBackward", backwardCheckBox->isChecked());
settings.endGroup();
```

La *valeur* peut être de type `int`, `bool`, `double`, `QString`, `QStringList`, ou de n'importe quel autre type pris en charge par `QVariant`, y compris des types personnalisés enregistrés.

```
void MainWindow::readSettings()
{
    QSettings settings("Software Inc.", "Spreadsheet");

    QRect rect = settings.value("geometry",
                                QRect(200, 200, 400, 400)).toRect();
    move(rect.topLeft());
    resize(rect.size());

    recentFiles = settings.value("recentFiles").toStringList();
    updateRecentFileActions();

    bool showGrid = settings.value("showGrid", true).toBool();
    showGridAction->setChecked(showGrid);

    bool autoRecalc = settings.value("autoRecalc", true).toBool();
    autoRecalcAction->setChecked(autoRecalc);
}
```

La fonction `readSettings()` charge les paramètres qui étaient sauvegardés par `writeSettings()`. Le deuxième argument de la fonction `value()` indique une valeur par défaut, dans le cas où aucun paramètre n'est disponible. Les valeurs par défaut sont utilisées la première fois que l'application est exécutée. Etant donné qu'aucun second argument n'est indiqué pour la liste des fichiers récents, il sera défini en liste vide à la première exécution.

Qt propose une fonction `QWidget::setGeometry()` pour compléter `QWidget::geometry()`, mais elle ne fonctionne pas toujours comme prévu sous X11 en raison des limites de la plupart des gestionnaires de fenêtre. C'est pour cette raison que nous utilisons plutôt `move()` et `resize()`. (Voir **http://doc.trolltech.com/4.1/geometry.html** pour une explication plus détaillée.)

Nous avons opté pour une organisation dans `MainWindow` parmi de nombreuses approches possibles, avec tout le code associé à `QSettings` dans `readSettings()` et `writeSettings()`. Un objet `QSettings` peut être créé pour identifier ou modifier un paramètre pendant l'exécution de l'application et n'importe où dans le code.

Nous avons désormais implémenté `MainWindow` dans `Spreadsheet`. Dans les sections suivantes, nous verrons comment l'application Spreadsheet peut être modifiée de manière à gérer plusieurs documents, et comment implémenter une page d'accueil. Nous compléterons ses fonctionnalités, notamment avec la gestion des formules et le tri, dans le prochain chapitre.

Documents multiples

Nous sommes désormais prêts à coder la fonction `main()` de l'application Spreadsheet :

```
#include <QApplication>

#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow mainWin;
    mainWin.show();
    return app.exec();
}
```

Cette fonction `main()` est légèrement différente de celles que nous avons écrites jusque là : nous avons créé l'instance `MainWindow` comme une variable sur la pile au lieu d'utiliser `new`. L'instance `MainWindow` est ensuite automatiquement détruite quand la fonction se termine.

Avec la fonction `main()` présentée ci-dessus, l'application Spreadsheet propose une seule fenêtre principale et ne peut gérer qu'un document à la fois. Si vous voulez modifier plusieurs documents en même temps, vous pourriez démarrer plusieurs instances de l'application Spreadsheet. Mais ce n'est pas aussi pratique pour les utilisateurs que d'avoir une seule instance de l'application proposant plusieurs fenêtres principales, tout comme une instance d'un navigateur Web peut fournir plusieurs fenêtres de navigateur simultanément.

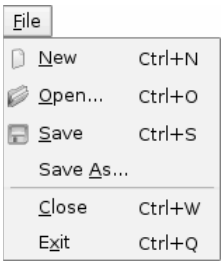
Nous modifierons l'application Spreadsheet de sorte qu'elle puisse gérer plusieurs documents. Nous avons tout d'abord besoin d'un menu File légèrement différent :

- File > New crée une nouvelle fenêtre principale avec un document vide, au lieu de réutiliser la fenêtre principale existante.

- File > Close ferme la fenêtre principale active.
- File > Exit ferme toutes les fenêtres.

Dans la version originale du menu File, il n'y avait pas d'option Close parce qu'elle aurait eu la même fonction qu'Exit.

Figure 3.16
Le nouveau menu File



Voici la nouvelle fonction `main()` :

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow *mainWin = new MainWindow;
    mainWin->show();
    return app.exec();
}
```

Avec plusieurs fenêtres, il est maintenant intéressant de créer `MainWindow` avec `new`, puisque nous avons ensuite la possibilité d'exécuter `delete` sur une fenêtre principale quand nous avons fini afin de libérer la mémoire.

Voici le nouveau slot `MainWindow::newFile()` :

```
void MainWindow::newFile()
{
    MainWindow *mainWin = new MainWindow;
    mainWin->show();
}
```

Nous créons simplement une nouvelle instance de `MainWindow`. Cela peut sembler stupide de ne pas conserver un pointeur vers la nouvelle fenêtre, mais ce n'est pas un problème étant donné que Qt assure le suivi de toutes les fenêtres pour nous.

Voici les actions pour Close et Exit :

```
void MainWindow::createActions()
{
   ...
   closeAction = new QAction(tr("&Close"), this);
   closeAction->setShortcut(tr("Ctrl+W"));
   closeAction->setStatusTip(tr("Close this window"));
```

```
    connect(closeAction, SIGNAL(triggered()), this, SLOT(close()));

    exitAction = new QAction(tr("E&xit"), this);
    exitAction->setShortcut(tr("Ctrl+Q"));
    exitAction->setStatusTip(tr("Exit the application"));
    connect(exitAction, SIGNAL(triggered()),
            qApp, SLOT(closeAllWindows()));
    ...
}
```

Le slot `QApplication::closeAllWindows()` ferme toutes les fenêtres de l'application, à moins qu'une d'elles ne refuse l'événement `close`. C'est exactement le comportement dont nous avons besoin ici. Nous n'avons pas à nous soucier des modifications non sauvegardées parce que `MainWindow::closeEvent()` s'en charge dès qu'une fenêtre est fermée.

Il semble que notre application est maintenant capable de gérer plusieurs fenêtres. Malheureusement, il reste un problème masqué : si l'utilisateur continue à créer et fermer des fenêtres principales, la machine pourrait éventuellement manquer de mémoire. C'est parce que nous continuons à créer des widgets `MainWindow` dans `newFile()`, sans jamais les effacer. Quand l'utilisateur ferme une fenêtre principale, le comportement par défaut consiste à la masquer, elle reste donc en mémoire. Vous risquez donc de rencontrer des problèmes si le nombre de fenêtres principales est important.

La solution est de définir l'attribut `Qt::WA_DeleteOnClose` dans le constructeur :

```
MainWindow::MainWindow()
{
    ...
    setAttribute(Qt::WA_DeleteOnClose);
    ...
}
```

Il ordonne à Qt de supprimer la fenêtre lorsqu'elle est fermée. L'attribut `Qt::WA_DeleteOnClose` est l'un des nombreux indicateurs qui peuvent être définis sur un `QWidget` pour influencer son comportement.

La fuite de mémoire n'est pas le seul problème rencontré. La conception de notre application d'origine supposait que nous aurions une seule fenêtre principale. Dans le cas de plusieurs fenêtres, chaque fenêtre principale possède sa propre liste de fichiers ouverts récemment et ses propres options. Il est évident que la liste des fichiers ouverts récemment doit être globale à toute l'application. En fait, il suffit de déclarer la variable `recentFiles` comme statique pour qu'une seule instance soit gérée par l'application. Mais nous devons ensuite garantir que tous les appels de `updateRecentFileActions()` destinés à mettre à jour le menu File concernent bien toutes les fenêtres principales. Voici le code pour obtenir ce résultat :

```
foreach (QWidget *win, QApplication::topLevelWidgets()) {
    if (MainWindow *mainWin = qobject_cast<MainWindow *>(win))
        mainWin->updateRecentFileActions();
}
```

Ce code s'appuie sur la construction `foreach` de Qt (expliquée au Chapitre 11) pour parcourir toutes les fenêtres de l'application et appelle `updateRecentFileActions()` sur tous les widgets de type `MainWindow`. Un code similaire peut être employé pour synchroniser les options `Show Grid` et `Auto-Recalculate` ou pour s'assurer que le même fichier n'est pas chargé deux fois.

Les applications qui proposent un document par fenêtre principale sont appelées des applications SDI (*single document interface*). Il existe une alternative courante sous Windows : MDI (*multiple document interface*), où l'application comporte une seule fenêtre principale qui gère plusieurs fenêtres de document dans sa zone d'affichage centrale. Qt peut être utilisé pour créer des applications SDI et MDI sur toutes les plates-formes prises en charge. La Figure 3.17 montre les deux versions de l'application Spreadsheet. MDI est abordé au Chapitre 6.

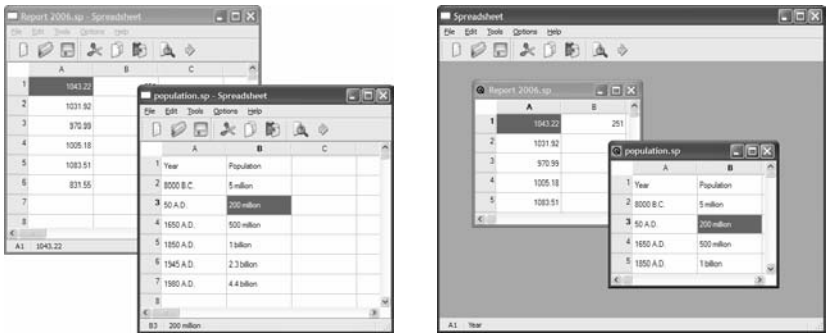


Figure 3.17
SDI versus MDI

Pages d'accueil

De nombreuses applications affichent une page d'accueil au démarrage. Certains développeurs se servent de cette page pour dissimuler un démarrage lent, alors que d'autres l'exploitent pour leurs services marketing. La classe `QSplashScreen` facilite l'ajout d'une page d'accueil aux applications Qt.

Cette classe affiche une image avant l'apparition de la fenêtre principale. Elle peut aussi écrire des messages sur l'image pour informer l'utilisateur de la progression du processus d'initialisation de l'application. En général, le code de la page d'accueil se situe dans `main()`, avant l'appel de `QApplication::exec()`.

Le code suivant est un exemple de fonction `main()` qui utilise `QSplashScreen` pour présenter une page d'accueil dans une application qui charge des modules et établit des connexions réseau au démarrage.

```
int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
```

```
    QSplashScreen *splash = new QSplashScreen;
    splash->setPixmap(QPixmap(":/images/splash.png"));
    splash->show();

    Qt::Alignment topRight = Qt::AlignRight | Qt::AlignTop;

    splash->showMessage(QObject::tr("Setting up the main window..."),
                        topRight, Qt::white);
    MainWindow mainWin;

    splash->showMessage(QObject::tr("Loading modules..."),
                        topRight, Qt::white);
    loadModules();

    splash->showMessage(QObject::tr("Establishing connections..."),
                        topRight, Qt::white);
    establishConnections();

    mainWin.show();
    splash->finish(&mainWin);
    delete splash;

    return app.exec();
}
```

Figure 3.18
Une page d'accueil

Nous avons désormais terminé l'étude de l'interface utilisateur de l'application Spreadsheet. Dans le prochain chapitre, vous compléterez l'application en implémentant la fonctionnalité principale du tableur.

4

Implémenter la fonctionnalité d'application

Au sommaire de ce chapitre

- ✔ Le widget central
- ✔ Dérivation de QTable
- ✔Widget
- ✔ Chargement et sauvegarde
- ✔ Implémenter le menu Edit
- ✔ Implémenter les autres menus
- ✔ Dérivation de QTableWidgetItem

Dans les deux précédents chapitres, nous vous avons expliqué comment créer l'interface utilisateur de l'application Spreadsheet. Dans ce chapitre, nous terminerons le programme en codant sa fonctionnalité sous-jacente. Nous verrons entre autres comment charger et sauvegarder des fichiers, stocker des données en mémoire, implémenter des opérations du presse-papiers et ajouter une prise en charge des formules de la feuille de calcul à `QTableWidget`.

Le widget central

La zone centrale d'un `QMainWindow` peut être occupée par n'importe quel type de widget. Voici quelques possibilités :

1. **Utiliser un widget Qt standard.**

   Un widget standard comme `QTableWidget` ou `QTextEdit` peut être employé comme widget central. Dans ce cas, la fonctionnalité de l'application, telle que le chargement et la sauvegarde des fichiers, doit être implémentée quelque part (par exemple dans une sous-classe `QMainWindow`).

2. **Utiliser un widget personnalisé.**

   Des applications spécialisées ont souvent besoin d'afficher des données dans un widget personnalisé. Par exemple, un programme d'éditeur d'icônes aurait un widget `IconEditor` comme widget central. Le Chapitre 5 vous explique comment écrire des widgets personnalisés dans Qt.

3. **Utiliser un `QWidget` ordinaire avec un gestionnaire de disposition.**

   Il peut arriver que la zone centrale de l'application soit occupée par plusieurs widgets. C'est possible grâce à l'utilisation d'un `QWidget` comme parent de tous les autres widgets et de gestionnaires de disposition pour dimensionner et positionner les widgets enfants.

4. **Utiliser un séparateur.**

   Il existe un autre moyen d'utiliser plusieurs widgets ensembles : un `QSplitter`. `QSplitter` dispose ses widgets enfants horizontalement ou verticalement et le séparateur offre la possibilité à l'utilisateur d'agir sur cette disposition. Les séparateurs peuvent contenir tout type de widgets, y compris d'autres séparateurs.

5. **Utiliser un espace de travail MDI.**

   Si l'application utilise MDI, la zone centrale est occupée par un widget `QWorkspace` et chaque fenêtre MDI est un enfant de ce widget.

Les dispositions, les séparateurs et les espaces de travail MDI peuvent être combinés à des widgets Qt standards ou personnalisés. Le Chapitre 6 traite de ces classes très en détail.

Concernant l'application Spreadsheet, une sous-classe `QTableWidget` sert de widget central. La classe `QTableWidget` propose déjà certaines fonctionnalités de feuille de calcul dont nous avons besoin, mais elle ne prend pas en charge les opérations du presse-papiers et ne comprend pas les formules comme "=A1+A2+A3".Nous implémenterons cette fonction manquante dans la classe `Spreadsheet`.

Dérivation de QTableWidget

La classe `Spreadsheet` hérite de `QTableWidget`. Un `QTableWidget` est une grille qui représente un tableau en deux dimensions. Il affiche n'importe quelle cellule que l'utilisateur fait défiler, dans ses dimensions spécifiées. Quand l'utilisateur saisit du texte dans une cellule vide, `QTableWidget` crée automatiquement un `QTableWidgetItem` pour stocker le texte.

Implémentons `Spreadsheet` en commençant par le fichier d'en-tête :

```
#ifndef SPREADSHEET_H
#define SPREADSHEET_H

#include <QTableWidget>

class Cell;
class SpreadsheetCompare;
```

L'en-tête commence par les déclarations préalables des classes `Cell` et `SpreadsheetCompare`.

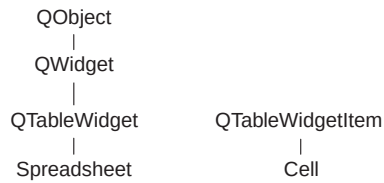


Figure 4.1
Arbres d'héritage pour Spreadsheet et Cell

Les attributs d'une cellule `QTableWidget`, tels que son texte et son alignement, sont conservés dans un `QTableWidgetItem`. Contrairement à `QTableWidget`, `QTableWidgetItem` n'est pas une classe de widget ; c'est une classe de données. La classe `Cell` hérite de `QTableWidgetItem`. Elle est détaillée pendant la présentation de son implémentation dans la dernière section de ce chapitre.

```
class Spreadsheet : public QTableWidget
{
   Q_OBJECT

public:
    Spreadsheet(QWidget *parent = 0);

    bool autoRecalculate() const { return autoRecalc; }
    QString currentLocation() const;
    QString currentFormula() const;
    QTableWidgetSelectionRange selectedRange() const;
    void clear();
    bool readFile(const QString &fileName);
    bool writeFile(const QString &fileName);
    void sort(const SpreadsheetCompare &compare);
```

La fonction `autoRecalculate()` est implémentée en mode inline (en ligne) parce qu'elle indique simplement si le recalcul automatique est activé ou non.

Dans le Chapitre 3, nous nous sommes basés sur des fonctions publiques dans `Spreadsheet` lorsque nous avons implémenté `MainWindow`. Par exemple, nous avons appelé `clear()` depuis `MainWindow::newFile()` pour réinitialiser la feuille de calcul. Nous avons aussi utilisé certaines fonctions héritées de `QTableWidget`, notamment `setCurrentCell()` et `setShowGrid()`.

```
public slots:
    void cut();
    void copy();
    void paste();
    void del();
    void selectCurrentRow();
    void selectCurrentColumn();
    void recalculate();
    void setAutoRecalculate(bool recalc);
    void findNext(const QString &str, Qt::CaseSensitivity cs);
    void findPrevious(const QString &str, Qt::CaseSensitivity cs);

signals:
    void modified();
```

`Spreadsheet` propose plusieurs slots implémentant des actions depuis les menus Edit, Tools et Options, de même qu'un signal, `modified()`, pour annoncer tout changement.

```
private slots:
    void somethingChanged();
```

Nous définissons un slot privé exploité en interne par la classe `Spreadsheet`.

```
private:
    enum { MagicNumber = 0x7F51C883, RowCount = 999, ColumnCount = 26 };
    Cell *cell(int row, int column) const;
    QString text(int row, int column) const;
    QString formula(int row, int column) const;
    void setFormula(int row, int column, const QString &formula);

    bool autoRecalc;
};
```

Dans la section privée de la classe, nous déclarons trois constantes, quatre fonctions et une variable.

```
class SpreadsheetCompare
{
public:
   bool operator()(const QStringList &row1,
                   const QStringList &row2) const;

   enum { KeyCount = 3 };
   int keys[KeyCount];
```

```
    bool ascending[KeyCount];
};

#endif
```

Le fichier d'en-tête se termine par la définition de la classe `SpreadsheetCompare`. Nous reviendrons sur ce point lorsque nous étudierons `Spreadsheet::sort()`.

Nous allons désormais passer en revue l'implémentation :

```
#include <QtGui>

#include "cell.h"
#include "spreadsheet.h"

Spreadsheet::Spreadsheet(QWidget *parent)
    : QTableWidget(parent)
{
    autoRecalc = true;

    setItemPrototype(new Cell);
    setSelectionMode(ContiguousSelection);

    connect(this, SIGNAL(itemChanged(QTableWidgetItem *)),
            this, SLOT(somethingChanged()));

    clear();
}
```

Normalement, quand l'utilisateur saisit du texte dans une cellule vide, `QTableWidget` crée automatiquement un `QTableWidgetItem` pour contenir le texte. Dans notre feuille de calcul, nous voulons plutôt créer des éléments `Cell`. Pour y parvenir, nous appelons `setItemPrototype()` dans le constructeur. En interne, `QTableWidget` copie l'élément transmis comme un prototype à chaque fois qu'un nouvel élément est requis.

Toujours dans le constructeur, nous définissons le mode de sélection en `QAbstractItemView::ContiguousSelection` pour autoriser une seule sélection rectangulaire. Nous connectons le signal `itemChanged()` du widget de la table au slot privé `somethingChanged()` ; lorsque l'utilisateur modifie une cellule, nous sommes donc sûrs que le slot `somethingChanged()` est appelé. Enfin, nous invoquons `clear()` pour redimensionner la table et configurer les en-têtes de colonne.

```
void Spreadsheet::clear()
{
    setRowCount(0);
    setColumnCount(0);
    setRowCount(RowCount);
    setColumnCount(ColumnCount);

    for (int i = 0; i < ColumnCount; ++i) {
        QTableWidgetItem *item = new QTableWidgetItem;
        item->setText(QString(QChar('A' + i)));
```

```
            setHorizontalHeaderItem(i, item);
        }

        setCurrentCell(0, 0);
    }
```

La fonction `clear()` est appelée depuis le constructeur `Spreadsheet` pour initialiser la feuille de calcul. Elle est aussi invoquée à partir de `MainWindow::newFile()`.

Nous aurions pu utiliser `QTableWidget::clear()` pour effacer tous les éléments et toutes les sélections, mais les en-têtes auraient conservé leurs tailles actuelles. Au lieu de cela, nous redimensionnons la table en 0 × 0. Toute la feuille de calcul est donc effacée, y compris les en-têtes. Nous redimensionnons ensuite la table en `ColumnCount _ RowCount` (26 _ 999) et nous alimentons l'en-tête horizontal avec des `QTableWidgetItem` qui contiennent les noms de colonne "A", "B", …, "Z".Nous ne sommes pas obligés de définir les intitulés des en-têtes verticaux, parce qu'ils présentent par défaut les valeurs suivantes : "1", "2",..., "999". Pour terminer, nous plaçons le curseur au niveau de la cellule A1.

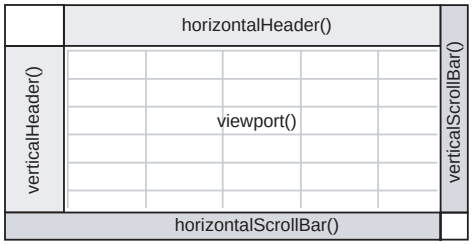


Figure 4.2
Les widgets qui constituent QTableWidget

Un `QTableWidget` se compose de plusieurs widgets enfants (voir Figure 4.2). Un `QHeaderView` horizontal est positionné en haut, un `QHeaderView` vertical à gauche et deux `QScrollBar` terminent cette composition. La zone au centre est occupée par un widget spécial appelé `viewport`, sur lequel notre `QTableWidget` dessine les cellules. Les divers widgets enfants sont accessibles par le biais de fonctions héritées de `QTableView` et `QAbstractScrollArea` (voir Figure 4.2). `QAbstractScrollArea` fournit un viewport équipé de deux barres de défilement, que vous pouvez activer ou désactiver. Sa sous-classe `QScrollArea` est abordée au Chapitre 6.

Stocker des données en tant qu'éléments

Dans l'application Spreadsheet, chaque cellule non vide est stockée en mémoire sous forme d'objet `QTableWidgetItem` individuel. Stocker des données en tant qu'éléments est une approche également utilisée par `QListWidget` et `QTreeWidget`, qui agissent sur `QListWidgetItem` et `QTreeWidgetItem`.

Les classes d'éléments de Qt peuvent être directement employées comme des conteneurs de données. Par exemple, un `QTableWidgetItem` stocke par définition quelques attributs, y compris une chaîne, une police, une couleur et une icône, ainsi qu'un pointeur vers `QTableWidget`. Les éléments ont aussi la possibilité de contenir des données (`QVariant`), dont des types personnalisés enregistrés, et en dérivant la classe d'éléments, nous pouvons proposer des fonctionnalités supplémentaires.

D'autres kits d'outils fournissent un pointeur `void` dans leurs classes d'éléments pour conserver des données personnalisées. Dans Qt, l'approche la plus naturelle consiste à utiliser `setData()` avec un `QVariant`, mais si un pointeur `void` est nécessaire, vous dériverez simplement une classe d'éléments et vous ajouterez une variable membre pointeur `void`.

Quand la gestion des données devient plus exigeante, comme dans le cas de jeux de données de grande taille, d'éléments de données complexes, d'une intégration de base de données et de vues multiples de données, Qt propose un ensemble de classes modèle/vue qui séparent les données de leur représentation visuelle. Ces thèmes sont traités au Chapitre 10.

```
Cell *Spreadsheet::cell(int row, int column) const
{
   return static_cast<Cell *>(item(row, column));
}
```

La fonction privée `cell()` retourne l'objet `Cell` pour une ligne et une colonne données. Elle est presque identique à `QTableWidget::item()`, sauf qu'elle renvoie un pointeur de `Cell` au lieu d'un pointeur de `QTableWidgetItem`.

```
QString Spreadsheet::text(int row, int column) const
{
   Cell *c = cell(row, column);
   if (c) {
      return c->text();
   } else {
      return "";
   }
}
```

La fonction privée `text()` retourne le texte d'une cellule particulière. Si `cell()` retourne un pointeur nul, la cellule est vide, une chaîne vide est donc renvoyée.

```
QString Spreadsheet::formula(int row, int column) const
{
   Cell *c = cell(row, column);
   if (c) {
      return c->formula();
   } else {
      return "";
   }
}
```

La fonction `formula()` retourne la formule de la cellule. Dans la plupart des cas, la formule et le texte sont identiques ; par exemple, la formule "Hello" détermine la chaîne "Hello", donc si l'utilisateur tape "Hello" dans une cellule et appuie sur Entrée, cette cellule affichera le texte "Hello".

Mais il y a quelques exceptions :

- Si la formule est un nombre, elle est interprétée en tant que tel. Par exemple, la formule "1,50" interprète la valeur en type `double` 1,5, qui est affiché sous la forme "1,5" justifié à droite dans la feuille de calcul.
- Si la formule commence par une apostrophe, le reste de la formule est considéré comme du texte. Par exemple, la formule " '12345" interprète cette valeur comme la chaîne "12345."
- Si la formule commence par un signe égal (=), elle est considérée comme une formule arithmétique. Par exemple, si la cellule A1 contient "12" et si la cellule A2 comporte le chiffre "6", la formule "=A1+A2" est égale à 18.

La tâche qui consiste à convertir une formule en valeur est accomplie par la classe `Cell`. Pour l'instant, l'important est de se souvenir que le texte affiché dans la cellule est le résultat de l'évaluation d'une formule et pas la formule en elle-même.

```
void Spreadsheet::setFormula(int row, int column,
                             const QString &formula)
{
   Cell *c = cell(row, column);
   if (!c) {
       c = new Cell;
       setItem(row, column, c);
   }
   c->setFormula(formula);
}
```

La fonction privée `setFormula()` définit la formule d'une cellule donnée. Si la cellule contient déjà un objet `Cell`, nous le réutilisons. Sinon, nous créons un nouvel objet `Cell` et nous appelons `QTableWidget::setItem()` pour l'insérer dans la table. Pour terminer, nous invoquons la propre fonction `setFormula()` de la cellule, pour actualiser cette dernière si elle est affichée à l'écran. Nous n'avons pas à nous soucier de supprimer l'objet `Cell` par la suite ; `QTableWidget` prend en charge la cellule et la supprimera automatiquement au moment voulu.

```
QString Spreadsheet::currentLocation() const
{
    return QChar('A' + currentColumn())
           + QString::number(currentRow() + 1);
}
```

La fonction `currentLocation()` retourne l'emplacement de la cellule actuelle dans le format habituel de la feuille de calcul, soit la lettre de la colonne suivie du numéro de la ligne. `MainWindow::updateStatusBar()` l'utilise pour afficher l'emplacement dans la barre d'état.

```
QString Spreadsheet::currentFormula() const
{
    return formula(currentRow(), currentColumn());
}
```

La fonction `currentFormula()` retourne la formule de la cellule en cours. Elle est invoquée à partir de `MainWindow::updateStatusBar()`.

```
void Spreadsheet::somethingChanged()
{
    if (autoRecalc)
        recalculate();
    emit modified();
}
```

Le slot privé `somethingChanged()` recalcule l'ensemble de la feuille de calcul si l'option de "recalcul automatique" est activée. Il émet également le signal `modified()`.

Chargement et sauvegarde

Nous allons désormais implémenter le chargement et la sauvegarde des fichiers `Spreadsheet` grâce à un format binaire personnalisé. Pour ce faire, nous emploierons `QFile` et `QDataStream` qui, ensemble, fournissent des entrées/sorties binaires indépendantes de la plate-forme.

Nous commençons par écrire un fichier Spreadsheet :

```
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QIODevice::WriteOnly)) {
        QMessageBox::warning(this, tr("Spreadsheet"),
                             tr("Cannot write file %1:\n%2.")
                             .arg(file.fileName())
                             .arg(file.errorString()));
        return false;
    }

    QDataStream out(&file);
    out.setVersion(QDataStream::Qt_4_1);

    out << quint32(MagicNumber);

    QApplication::setOverrideCursor(Qt::WaitCursor);
    for (int row = 0; row < RowCount; ++row) {
        for (int column = 0; column < ColumnCount; ++column) {
            QString str = formula(row, column);
            if (!str.isEmpty())
                out << quint16(row) << quint16(column) << str;
        }
    }
    QApplication::restoreOverrideCursor();
    return true;
}
```

La fonction `writeFile()` est appelée depuis `MainWindow::saveFile()` pour écrire le fichier sur le disque. Elle retourne `true` en cas de succès et `false` en cas d'erreur.

Nous créons un objet `QFile` avec un nom de fichier donné et nous invoquons `open()` pour ouvrir le fichier en écriture. Nous créons aussi un objet `QDataStream` qui agit sur le `QFile` et s'en sert pour écrire les données.

Juste avant d'écrire les données, nous changeons le pointeur de l'application en pointeur d'attente standard (généralement un sablier) et nous restaurons le pointeur normal lorsque toutes les données ont été écrites. A la fin de la fonction, le fichier est fermé automatiquement par le destructeur de `QFile`.

`QDataStream` prend en charge les types C++ de base, de même que plusieurs types de Qt. La syntaxe est conçue selon les classes `<iostream>` du langage C++ Standard. Par exemple,

```
out << x << y << z;
```

écrit les variables `x`, `y` et `z` dans un flux, et

```
in >> x >> y >> z;
```

les lit depuis un flux. Etant donné que les types C++ de base `char`, `short`, `int`, `long` et `long long` peuvent présenter des tailles différentes selon les plates-formes, il est plus sûr de convertir ces valeurs en `qint8`, `quint8`, `qint16`, `quint16`, `qint32`, `quint32`, `qint64` ou `quint64` ; vous avez ainsi la garantie de travailler avec un type de la taille annoncée (en bits).

Le format de fichier de l'application Spreadsheet est assez simple (voir Figure 4.3). Un fichier Spreadsheet commence par un numéro sur 32 bits qui identifie le format de fichier (`MagicNumber`, défini par `0x7F51C883` dans `spreadsheet.h`, un chiffre aléatoire). Ce numéro est suivi d'une série de blocs, chacun d'eux contenant la ligne, la colonne et la formule d'une seule cellule. Pour économiser de l'espace, nous n'écrivons pas de cellules vides.

Figure 4.3
Le format du fichier Spreadsheet

La représentation binaire précise des types de données est déterminée par `QDataStream`. Par exemple, un type `quint16` est stocké sous forme de deux octets en ordre big-endian, et un `QString` se compose de la longueur de la chaîne suivie des caractères Unicode.

La représentation binaire des types Qt a largement évolué depuis Qt 1.0. Il est fort probable qu'elle continue son développement dans les futures versions de Qt pour suivre l'évolution des types existants et pour tenir compte des nouveaux types Qt. Par défaut, `QDataStream` utilise la version la plus récente du format binaire (version 7 dans Qt 4.1), mais il peut être configuré de manière à lire des versions antérieures. Pour éviter tout problème de compatibilité si l'application est recompilée par la suite avec une nouvelle version de Qt, nous demandons explicitement à `QDataStream` d'employer la version 7 quelle que soit la version de Qt utilisée pour la compilation. (`QDataStream::Qt_4_1` est une constante pratique qui vaut 7.)

`QDataStream` est très polyvalent. Il peut être employé sur `QFile`, mais aussi sur `QBuffer`, `QProcess`, `QTcpSocket` ou `QUdpSocket`. Qt propose également une classe `QTextStream` qui

peut être utilisée à la place de `QDataStream` pour lire et écrire des fichiers texte. Le Chapitre 12 se penche en détail sur ces classes et décrit les diverses approches consistant à gérer les différentes versions de `QDataStream`.

```
bool Spreadsheet::readFile(const QString &fileName)
{
   QFile file(fileName);
   if (!file.open(QIODevice::ReadOnly)) {
       QMessageBox::warning(this, tr("Spreadsheet"),
                            tr("Cannot read file %1:\n%2.")
                            .arg(file.fileName())
                            .arg(file.errorString()));
       return false;
   }

   QDataStream in(&file);
   in.setVersion(QDataStream::Qt_4_1);

   quint32 magic;
   in >> magic;
   if (magic != MagicNumber) {
       QMessageBox::warning(this, tr("Spreadsheet"),
                            tr("The file is not a Spreadsheet file."));
       return false;
   }

   clear();

   quint16 row;
   quint16 column;
   QString str;

   QApplication::setOverrideCursor(Qt::WaitCursor);
   while (!in.atEnd()) {
       in >> row >> column >> str;
       setFormula(row, column, str);
   }
   QApplication::restoreOverrideCursor();
   return true;
}
```

La fonction `readFile()` ressemble beaucoup à `writeFile()`. Nous utilisons `QFile` pour lire le fichier, mais cette fois-ci avec `QIODevice::ReadOnly` et pas `QIODevice::WriteOnly`. Puis nous définissons la version de `QDataStream` en 7. Le format de lecture doit toujours être le même que celui de l'écriture.

Si le fichier débute par le nombre magique approprié, nous appelons `clear()` pour vider toutes les cellules de la feuille de calcul et nous lisons les données de la cellule. Vu que le fichier ne contient que des données pour des cellules non vides, et qu'il est très improbable que chaque cellule de la feuille de calcul soit définie, nous devons nous assurer que toutes les cellules sont effacées avant la lecture.

Implémenter le menu Edit

Nous sommes désormais prêt à implémenter les slots qui correspondent au menu Edit de l'application. Ce menu est présenté en Figure 4.4.

```
void Spreadsheet::cut()
{
    copy();
    del();
}
```

Le slot `cut()` correspond à Edit > Cut. L'implémentation est simple parce que Cut est équivalent à Copy suivi de Delete.

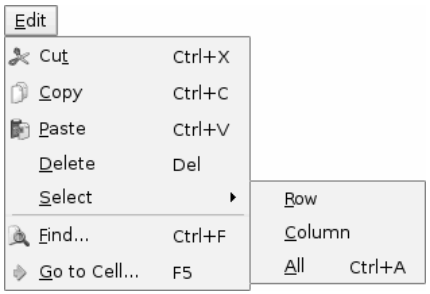


Figure 4.4
Le menu Edit de l'application Spreadsheet

```
void Spreadsheet::copy()
{
   QTableWidgetSelectionRange range = selectedRange();
   QString str;

   for (int i = 0; i < range.rowCount(); ++i) {
       if (i > 0)
           str += "\n";
       for (int j = 0; j < range.columnCount(); ++j) {
           if (j > 0)
               str += "\t";
           str += formula(range.topRow() + i, range.leftColumn() + j);
      }
   }
   QApplication::clipboard()->setText(str);
}
```

Le slot `copy()` correspond à Edit > Copy. Il parcourt la sélection actuelle (qui est simplement la cellule en cours s'il n'y a pas de sélection explicite). La formule de chaque cellule sélectionnée est ajoutée à `QString`, avec des lignes séparées par des sauts de ligne et des colonnes séparées par des tabulations.

Le presse-papiers est disponible dans Qt par le biais de la fonction statique `QApplication::clipboard()`. En appelant `QClipboard::setText()`, le texte est disponible dans le

presse-papiers, à la fois pour cette application et pour d'autres qui prennent en charge le texte brut (voir Figure 4.5). Notre format, basé sur les tabulations et les sauts de lignes en tant que séparateurs, est compris par une multitude d'applications, dont Microsoft Excel.

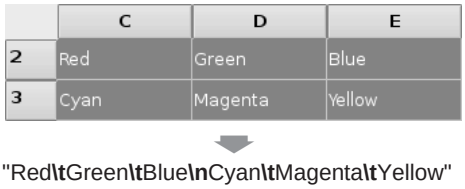


Figure 4.5
Copier une sélection dans le presse-papiers

La fonction `QTableWidget::selectedRanges()` retourne une liste de plages de sélection. Nous savons qu'il ne peut pas y en avoir plus d'une, parce que nous avons défini le mode de sélection en `QAbstractItemView::ContiguousSelection` dans le constructeur. Par souci de commodité, nous configurons une fonction `selectedRange()` qui retourne la plage de sélection :

```
QTableWidgetSelectionRange Spreadsheet::selectedRange() const
{
    QList<QTableWidgetSelectionRange> ranges = selectedRanges();
    if (ranges.isEmpty())
        return QTableWidgetSelectionRange();
    return ranges.first();
}
```

S'il n'y a qu'une sélection, nous retournons simplement la première (et unique). Vous ne devriez jamais vous trouver dans le cas où il n'y a aucune sélection, étant donné que le mode `ContiguousSelection` considère la cellule en cours comme sélectionnée. Toutefois pour prévenir tout bogue dans notre programme, nous gérons ce cas de figure.

```
void Spreadsheet::paste()
{
    QTableWidgetSelectionRange range = selectedRange();
    QString str = QApplication::clipboard()->text();
    QStringList rows = str.split('\n');
    int numRows = rows.count();
    int numColumns = rows.first().count('\t') + 1;

    if (range.rowCount() * range.columnCount() != 1
            && (range.rowCount() != numRows
                || range.columnCount() != numColumns)) {
        QMessageBox::information(this, tr("Spreadsheet"),
                tr("The information cannot be pasted because the copy "
                   "and paste areas aren't the same size."));
        return;
    }

    for (int i = 0; i < numRows; ++i) {
```

```
        QStringList columns = rows[i].split('\t');
        for (int j = 0; j < numColumns; ++j) {
             int row = range.topRow() + i;
             int column = range.leftColumn() + j;
             if (row < RowCount && column < ColumnCount)
                 setFormula(row, column, columns[j]);
        }
    }
    somethingChanged();
}
```

Le slot `paste()` correspond à Edit > Paste. Nous récupérons le texte dans le presse-papiers et nous appelons la fonction statique `QString::split()` pour adapter la chaîne au `QStringList`. Chaque ligne devient une chaîne dans la liste.

Nous déterminons ensuite la dimension de la zone de copie. Le nombre de lignes correspond au nombre de chaînes dans `QStringList` ; le nombre de colonnes est le nombre de tabulations à la première ligne, plus 1. Si une seule cellule est sélectionnée, nous nous servons de cette cellule comme coin supérieur gauche de la zone de collage ; sinon, nous utilisons la sélection actuelle comme zone de collage.

Pour effectuer le collage, nous parcourons les lignes et nous les divisons en cellules grâce à `QString::split()`, mais cette fois-ci avec une tabulation comme séparateur. La Figure 4.6 illustre ces étapes.

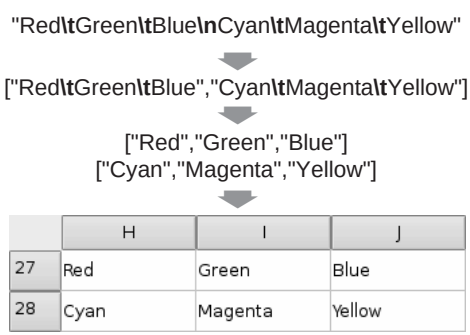


Figure 4.6
Coller le texte du presse-papiers dans la feuille de calcul

```
void Spreadsheet::del()
{
    foreach (QTableWidgetItem *item, selectedItems())
        delete item;
}
```

Le slot `del()` correspond à Edit > Delete. Il suffit d'utiliser `delete` sur chaque objet `Cell` de la sélection pour effacer les cellules. `QTableWidget` remarque quand ses `QTableWidgetItem` sont supprimés et se redessine automatiquement si l'un des éléments était visible. Si nous invoquons `cell()` avec l'emplacement d'une cellule supprimée, il renverra un pointeur nul.

```
void Spreadsheet::selectCurrentRow()
{
    selectRow(currentRow());
}

void Spreadsheet::selectCurrentColumn()
{
    selectColumn(currentColumn());
}
```

Les fonctions `selectCurrentRow()` et `selectCurrentColumn()` correspondent aux options Edit > Select > Row et Edit > Select > Column. Les implémentations reposent sur les fonctions `selectRow()` et `selectColumn()` de `QTableWidget`. Nous n'avons pas à implémenter la fonctionnalité correspondant à Edit > Select > All, étant donné qu'elle est proposée par la fonction héritée `QAbstractItemView::selectAll()` de `QTableWidget`.

```
void Spreadsheet::findNext(const QString &str, Qt::CaseSensitivity cs)
{
    int row = currentRow();
    int column = currentColumn() + 1;

    while (row < RowCount) {
       while (column < ColumnCount) {
          if (text(row, column).contains(str, cs)) {
              clearSelection();
              setCurrentCell(row, column);
              activateWindow();
              return;
          }
          ++column;
       }
       column = 0;
       ++row;
   }
   QApplication::beep();
}
```

Le slot `findNext()` parcourt les cellules en commençant par la cellule à droite du pointeur et en se dirigeant vers la droite jusqu'à la dernière colonne, puis il poursuit par la première colonne dans la ligne du dessous et ainsi de suite jusqu'à trouver le texte recherché ou jusqu'à atteindre la toute dernière cellule. Par exemple, si la cellule en cours est la cellule C24, nous recherchons D24, E24, …, Z24, puis A25, B25, C25, …, Z25, et ainsi de suite jusqu'à Z999. Si nous trouvons une correspondance, nous supprimons la sélection actuelle, nous déplaçons le pointeur vers cette cellule et nous activons la fenêtre qui contient `Spreadsheet`. Si aucune correspondance n'est découverte, l'application émet un signal sonore pour indiquer que la recherche n'a pas abouti.

```
void Spreadsheet::findPrevious(const QString &str,
                               Qt::CaseSensitivity cs)
{
```

```
    int row = currentRow();
    int column = currentColumn() - 1;

    while (row >= 0) {
       while (column >= 0) {
          if (text(row, column).contains(str, cs)) {
              clearSelection();
              setCurrentCell(row, column);
              activateWindow();
              return;
          }
          --column;
       }
       column = ColumnCount - 1;
       --row;
    }
    QApplication::beep();
}
```

Le slot `findPrevious()` est similaire à `findNext()`, sauf qu'il effectue une recherche dans l'autre sens et s'arrête à la cellule A1.

Implémenter les autres menus

Nous allons maintenant implémenter les slots des menus Tools et Options. Ces menus sont illustrés en Figure 4.7.

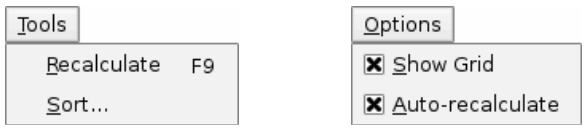


Figure 4.7
Les menus Tools et Options de l'application Spreadsheet

```
void Spreadsheet::recalculate()
{
   for (int row = 0; row < RowCount; ++row) {
       for (int column = 0; column < ColumnCount; ++column) {
           if (cell(row, column))
               cell(row, column)->setDirty();
       }
   }
   viewport()->update();
}
```

Le slot `recalculate()` correspond à Tools > Recalculate. Il est aussi appelé automatiquement par `Spreadsheet` si nécessaire.

Nous parcourons toutes les cellules et invoquons `setDirty()` sur chacune d'elles pour signaler celles qui doivent être recalculées. La prochaine fois que `QTableWidget` appelle `text()` sur `Cell` pour obtenir la valeur à afficher dans la feuille de calcul, la valeur sera recalculée.

Nous appelons ensuite `update()` sur le viewport pour redessiner la feuille de calcul complète. Le code de réaffichage dans `QTableWidget` invoque ensuite `text()` sur chaque cellule visible pour obtenir la valeur à afficher. Vu que nous avons appelé `setDirty()` sur chaque cellule, les appels de `text()` utiliseront une valeur nouvellement calculée. Le calcul pourrait exiger que les cellules non visibles soient recalculées, répercutant la même opération jusqu'à ce que chaque cellule qui a besoin d'être recalculée pour afficher le bon texte dans le viewport ait une valeur réactualisée. Le calcul est effectué par la classe `Cell`.

```
void Spreadsheet::setAutoRecalculate(bool recalc)
{
   autoRecalc = recalc;
   if (autoRecalc)
       recalculate();
}
```

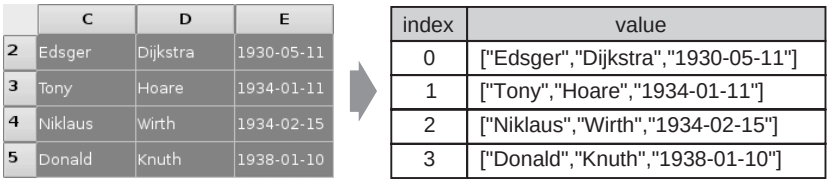
Le slot `setAutoRecalculate()` correspond à Options > Auto-Recalculate. Si la fonction est activée, nous recalculons immédiatement la feuille de calcul pour nous assurer qu'elle est à jour ; ensuite, `recalculate()` est appelé automatiquement dans `somethingChanged()`.

Nous n'avons pas besoin d'implémenter quoi que ce soit pour Options > Show Grid, parce que `QTableWidget` a déjà un slot `setShowGrid()` qu'il a hérité de sa classe de base `QTableView`. Tout ce qui reste, c'est `Spreadsheet::sort()` qui est invoquée dans `MainWindow::sort()` :

```
void Spreadsheet::sort(const SpreadsheetCompare &compare)
{
   QList<QStringList> rows;
   QTableWidgetSelectionRange range = selectedRange();
   int i;

   for (i = 0; i < range.rowCount(); ++i) {
       QStringList row;
       for (int j = 0; j < range.columnCount(); ++j)
           row.append(formula(range.topRow() + i,
                              range.leftColumn() + j));
       rows.append(row);
   }

   qStableSort(rows.begin(), rows.end(), compare);

   for (i = 0; i < range.rowCount(); ++i) {
       for (int j = 0; j < range.columnCount(); ++j)
           setFormula(range.topRow() + i, range.leftColumn() + j,
                      rows[i][j]);
   }

   clearSelection();
   somethingChanged();
}
```

Le tri s'opère sur la sélection actuelle et réorganise les lignes selon les clés et les ordres de tri stockés dans l'objet `compare`. Nous représentons chaque ligne de données avec `QStringList` et nous conservons la sélection sous forme de liste de lignes (voir Figure 4.8). Nous nous servons de l'algorithme `qStableSort()` de Qt et pour une question de simplicité, le tri s'effectue sur la formule plutôt que sur la valeur. Les algorithmes standards, de même que les structures de données de Qt sont abordés au Chapitre 11.

Figure 4.8
Stocker la sélection sous forme de liste de lignes

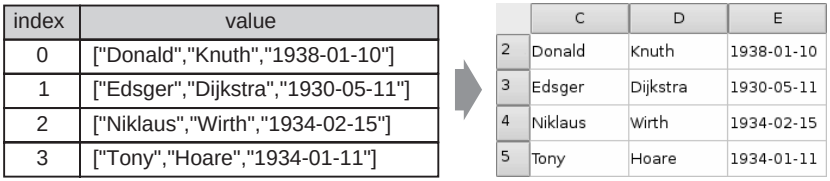


	C	D	E
2	Edsger	Dijkstra	1930-05-11
3	Tony	Hoare	1934-01-11
4	Niklaus	Wirth	1934-02-15
5	Donald	Knuth	1938-01-10

index	value
0	["Edsger","Dijkstra","1930-05-11"]
1	["Tony","Hoare","1934-01-11"]
2	["Niklaus","Wirth","1934-02-15"]
3	["Donald","Knuth","1938-01-10"]

La fonction `qStableSort()` reçoit un itérateur de début et de fin, ainsi qu'une fonction de comparaison. La fonction de comparaison est une fonction qui reçoit deux arguments (deux `QStringList`) et qui retourne `true` si le premier argument est "inférieur" au second argument et `false` dans les autres cas. L'objet `compare` que nous transmettons comme fonction de comparaison n'est pas vraiment une fonction, mais il peut être utilisé comme telle, comme nous allons le voir.

Figure 4.9
Réintégrer les données dans la table après le tri



index	value
0	["Donald","Knuth","1938-01-10"]
1	["Edsger","Dijkstra","1930-05-11"]
2	["Niklaus","Wirth","1934-02-15"]
3	["Tony","Hoare","1934-01-11"]

	C	D	E
2	Donald	Knuth	1938-01-10
3	Edsger	Dijkstra	1930-05-11
4	Niklaus	Wirth	1934-02-15
5	Tony	Hoare	1934-01-11

Après avoir exécuté `qStableSort()`, nous réintégrons les données dans la table (voir Figure 4.9), nous effaçons la sélection et nous appelons `somethingChanged()`.

Dans `spreadsheet.h`, la classe `SpreadsheetCompare` était définie comme suit :

```
class SpreadsheetCompare
{
public:
    bool operator()(const QStringList &row1,
                    const QStringList &row2) const;

    enum { KeyCount = 3 };
    int keys[KeyCount];
    bool ascending[KeyCount];
};
```

La classe `SpreadsheetCompare` est spéciale parce qu'elle implémente un opérateur (). Nous avons donc la possibilité d'utiliser la classe comme si c'était une fonction. De telles classes sont appelées des objets fonction, ou foncteurs. Pour comprendre comment fonctionnent les foncteurs, nous débutons par un exemple simple :

```
class Square
{
public:
    int operator()(int x) const { return x * x; }
}
```

La classe `Square` fournit une fonction, `operator()(int)`, qui retourne le carré de son paramètre. En nommant la fonction `operator()(int)`, au lieu de `compute(int)` par exemple, nous avons la possibilité d'utiliser un objet de type `Square` comme si c'était une fonction :

```
Square square;
int y = square(5);
```

A présent, analysons un exemple impliquant `SpreadsheetCompare` :

```
QStringList row1, row2;
QSpreadsheetCompare compare;
...
if (compare(row1, row2)) {
   // row1 est inférieure à
}
```

L'objet `compare` peut être employé comme s'il était une fonction `compare()` ordinaire. De plus, son implémentation peut accéder à toutes les clés et ordres de tri stockés comme variables membres.

Il existe une alternative : nous aurions pu conserver tous les ordres et clés de tri dans des variables globales et utiliser une fonction `compare()` ordinaire. Cependant, la communication *via* les variables globales n'est pas très élégante et peut engendrer des bogues subtils. Les foncteurs sont plus puissants pour interfacer avec des fonctions modèles comme `qStableSort()`.

Voici l'implémentation de la fonction employée pour comparer deux lignes de la feuille de calcul :

```
bool SpreadsheetCompare::operator()(const QStringList &row1,
                                    const QStringList &row2) const
{
    for (int i = 0; i < KeyCount; ++i) {
        int column = keys[i];
        if (column != -1) {
            if (row1[column] != row2[column]) {
                if (ascending[i]) {
                    return row1[column] < row2[column];
                } else {
                    return row1[column] > row2[column];
                }
            }
         }
    }
    return false;
}
```

L'opérateur retourne `true` si la première ligne est inférieure à la seconde et `false` dans les autres cas. La fonction `qStableSort()` utilise ce résultat pour effectuer le tri.

Les tables `keys` et `ascending` de l'objet `SpreadsheetCompare` sont alimentées dans la fonction `MainWindow::sort()` (vue au Chapitre 2). Chaque clé possède un index de colonne ou –1 ("None").

Nous comparons les entrées de cellules correspondantes dans les deux lignes pour chaque clé dans l'ordre. Dès que nous découvrons une différence, nous retournons une valeur `true` ou `false`. S'il s'avère que toutes les comparaisons sont égales, nous retournons `false`. La fonction `qStableSort()` s'appuie sur l'ordre avant le tri pour résoudre les situations d'égalité ; si `row1` précédait `row2` à l'origine et n'est jamais "inférieur à" l'autre, `row1` précédera toujours `row2` dans le résultat. C'est ce qui distingue `qStableSort()` de son cousin `qSort()` dont le résultat est moins prévisible.

Nous avons désormais terminé la classe `Spreadsheet`. Dans la prochaine section, nous allons analyser la classe `Cell`. Cette classe est employée pour contenir les formules des cellules et propose une réimplémentation de la fonction `QTableWidgetItem::data()` que `Spreadsheet` appelle indirectement par le biais de la fonction `QTableWidgetItem::text()`. L'objectif est d'afficher le résultat du calcul de la formule d'une cellule.

Dérivation de QTableWidgetItem

La classe `Cell` hérite de `QTableWidgetItem`. Cette classe est conçue pour bien fonctionner avec `Spreadsheet`, mais elle ne dépend pas spécifiquement de cette classe et pourrait, en théorie, être utilisée dans n'importe quel `QTableWidget`. Voici le fichier d'en-tête :

```
#ifndef CELL_H
#define CELL_H

#include <QTableWidgetItem>

class Cell : public QTableWidgetItem
{
public:
    Cell();

    QTableWidgetItem *clone() const;
    void setData(int role, const QVariant &value);
    QVariant data(int role) const;
    void setFormula(const QString &formula);
    QString formula() const;
    void setDirty();

private:
    QVariant value() const;
    QVariant evalExpression(const QString &str, int &pos) const;
    QVariant evalTerm(const QString &str, int &pos) const;
```

```
    QVariant evalFactor(const QString &str, int &pos) const;

    mutable QVariant cachedValue;
    mutable bool cacheIsDirty;
};

#endif
```

La classe `Cell` développe `QTableWidgetItem` en ajoutant deux variables privées :

- `cachedValue` met en cache la valeur de la cellule sous forme de `QVariant`.
- `cacheIsDirty` est `true` si la valeur mise en cache n'est pas à jour.

Nous utilisons `QVariant` parce que certaines cellules ont une valeur `double` alors que d'autres ont une valeur `QString`.

Les variables `cachedValue` et `cacheIsDirty` sont déclarées avec le mot-clé C++ `mutable`. Nous avons ainsi la possibilité de modifier ces variables dans des fonctions const. Nous pourrions aussi recalculer la valeur à chaque fois que `text()` est appelée, mais ce serait tout à fait inefficace.

Notez qu'il n'y a pas de macro `Q_OBJECT` dans la définition de classe. `Cell` est une classe C++ ordinaire, sans signaux ni slots. En fait, vu que `QTableWidgetItem` n'hérite pas de `QObject`, nous ne pouvons pas avoir de signaux et de slots dans `Cell`. Les classes d'éléments de Qt n'héritent pas de `QObject` pour optimiser les performances. Si des signaux et des slots se révèlent nécessaires, ils peuvent être implémentés dans le widget qui contient les éléments ou, exceptionnellement, en utilisant l'héritage multiple avec `QObject`.

Voici le début de `cell.cpp` :

```
#include <QtGui>

#include "cell.h"

Cell::Cell()
{
   setDirty();
}
```

Dans le constructeur, nous devons simplement définir le cache comme étant à actualiser (dirty). Vous n'avez pas besoin de transmettre un parent ; quand la cellule est insérée dans un `QTableWidget` avec `setItem()`, le `QTableWidget` prend automatiquement possession de celle-ci.

Chaque `QTableWidgetItem` peut contenir des données, jusqu'à un `QVariant` pour chaque "rôle" de données. Les rôles les plus couramment utilisés sont `Qt::EditRole` et `Qt::DisplayRole`. Le rôle de modification est employé pour des données qui doivent être modifiées et le rôle d'affichage pour des données qui doivent être affichées. Il arrive souvent que ces données soient les mêmes, mais dans `Cell` le rôle de modification correspond à la

formule de la cellule et le rôle d'affichage à la valeur de la cellule (le résultat de l'évaluation de la formule).

```
QTableWidgetItem *Cell::clone() const
{
    return new Cell(*this);
}
```

La fonction `clone()` est invoquée par `QTableWidget` quand il doit créer une nouvelle cellule – par exemple quand l'utilisateur commence à taper dans une cellule vide qui n'a encore jamais été utilisée. L'instance transmise à `QTableWidget::setItemPrototype()` est l'élément qui est cloné. Vu que la copie au niveau du membre est suffisante pour `Cell`, nous nous basons sur le constructeur de copie par défaut créé automatiquement par C++ dans le but de créer de nouvelles instances `Cell` dans la fonction `clone()`.

```
void Cell::setFormula(const QString &formula)
{
    setData(Qt::EditRole, formula);
}
```

La fonction `setFormula()` définit la formule de la cellule. C'est simplement une fonction pratique permettant d'appeler `setData()` avec le rôle de modification. Elle est invoquée dans `Spreadsheet::setFormula()`.

```
QString Cell::formula() const
{
    return data(Qt::EditRole).toString();
}
```

La fonction `formula()` est appelée dans `Spreadsheet::formula()`. Comme `setFormula()`, c'est une fonction commode, mais cette fois-ci qui récupère les données `EditRole` de l'élément.

```
void Cell::setData(int role, const QVariant &value)
{
    QTableWidgetItem::setData(role, value);
    if (role == Qt::EditRole)
        setDirty();
}
```

Si nous avons affaire à une nouvelle formule, nous définissons `cacheIsDirty` en `true` pour garantir que la cellule sera recalculée la prochaine fois que `text()` est appelé.

Aucune fonction `text()` n'est définie dans `Cell`, même si nous appelons `text()` sur des instances `Cell` dans `Spreadsheet::text()`. La fonction `text()` est une fonction de convenance proposée par `QTableWidgetItem` ; cela revient au même que d'appeler `data(Qt::DisplayRole).toString()`.

```
void Cell::setDirty()
{
   cacheIsDirty = true;
}
```

La fonction `setDirty()` est invoquée pour forcer le recalcul de la valeur d'une cellule. Elle définit simplement `cacheIsDirty` en `true`, ce qui signifie que `cachedValue` n'est plus à jour. Le recalcul n'est effectué que lorsqu'il est nécessaire.

```
QVariant Cell::data(int role) const
{
    if (role == Qt::DisplayRole) {
        if (value().isValid()) {
            return value().toString();
       } else {
           return "####";
       }
   } else if (role == Qt::TextAlignmentRole) {
       if (value().type() == QVariant::String) {
           return int(Qt::AlignLeft | Qt::AlignVCenter);
       } else {
          return int(Qt::AlignRight | Qt::AlignVCenter);
       }
   } else {
       return QTableWidgetItem::data(role);
   }
}
```

La fonction `data()` est réimplémentée dans `QTableWidgetItem`. Elle retourne le texte qui doit être affiché dans la feuille de calcul si elle est appelée avec `Qt::DisplayRole`, et la formule si elle est invoquée avec `Qt::EditRole`. Elle renvoie l'alignement approprié si elle est appelée avec `Qt::TextAlignmentRole`. Dans le cas de `DisplayRole`, elle se base sur `value()` pour calculer la valeur de la cellule. Si la valeur n'est pas valide (parce que la formule est mauvaise), nous retournons "####".

La fonction `Cell::value()` utilisée dans `data()` retourne un `QVariant`. Un `QVariant` peut stocker des valeurs de différents types, comme `double` et `QString`, et propose des fonctions pour convertir les variants dans d'autres types. Par exemple, appeler `toString()` sur un variant qui contient une valeur `double` produit une chaîne de `double`. Un `QVariant` construit avec le constructeur par défaut est un variant "invalide".

```
const QVariant Invalid;

QVariant Cell::value() const
{
    if (cacheIsDirty) {
        cacheIsDirty = false;

        QString formulaStr = formula();
        if (formulaStr.startsWith('\'')) {
            cachedValue = formulaStr.mid(1);
        } else if (formulaStr.startsWith('=')) {
            cachedValue = Invalid;
            QString expr = formulaStr.mid(1);
            expr.replace(" ", "");
            expr.append(QChar::Null);
```

```
            int pos = 0;
            cachedValue = evalExpression(expr, pos);
            if (expr[pos] != QChar::Null)
                cachedValue = Invalid;
        } else {
            bool ok;
            double d = formulaStr.toDouble(&ok);
            if (ok) {
                cachedValue = d;
            } else {
                cachedValue = formulaStr;
            }
         }
    }
    return cachedValue;
}
```

La fonction privée `value()` retourne la valeur de la cellule. Si `cacheIsDirty` est `true`, nous devons la recalculer.

Si la formule commence par une apostrophe (par exemple " '12345"), l'apostrophe se trouve à la position 0 et la valeur est la chaîne allant de la position 1 à la fin.

Si la formule commence par un signe égal (=), nous prenons la chaîne à partir de la position 1 et nous supprimons tout espace qu'elle contient. Nous appelons ensuite `evalExpression()` pour calculer la valeur de l'expression. L'argument `pos` est transmis par référence ; il indique la position du caractère où l'analyse doit commencer. Après l'appel de `evalExpression()`, le caractère à la position `pos` doit être le caractère `QChar::Null` que nous avons ajouté, s'il a été correctement analysé. Si l'analyse a échoué avant la fin, nous définissons `cachedValue` de sorte qu'il soit `Invalid`.

Si la formule ne commence pas par une apostrophe ni par un signe égal, nous essayons de la convertir en une valeur à virgule flottante à l'aide de `toDouble()`. Si la conversion fonctionne, nous configurons `cachedValue` pour y stocker le nombre obtenu ; sinon, nous définissons `cachedValue` avec la chaîne de la formule. Par exemple, avec une formule de "1,50," `toDouble()` définit `ok` en `true` et retourne 1,5, alors qu'avec une formule de "World Population" `toDouble()` définit `ok` en `false` et renvoie 0,0.

En transmettant à `toDouble()` un pointeur de type `bool`, nous sommes en mesure de faire une distinction entre la conversion d'une chaîne qui donne la valeur numérique 0,0 et une erreur de conversion (où 0,0 est aussi retourné mais `bool` est défini en `false`). Il est cependant parfois nécessaire de retourner une valeur zéro sur un échec de conversion, auquel cas nous n'avons pas besoin de transmettre de pointeur de `bool`. Pour des questions de performances et de portabilité, Qt n'utilise jamais d'exceptions C++ pour rapporter des échecs. Ceci ne vous empêche pas de les utiliser dans des programmes Qt, à condition que votre compilateur les prenne en charge.

La fonction `value()` est déclarée `const`. Nous devions déclarer `cachedValue` et `cacheIsValid` comme des variables mutable, de sorte que le compilateur nous permette de les modifier dans des fonctions const. Ce pourrait être tentant de rendre `value()` non-const et de supprimer

les mots clés `mutable`, mais le résultat ne compilerait pas parce que nous appelons `value()` depuis `data()`, une fonction const.

Nous avons désormais fini l'application Spreadsheet, excepté l'analyse des formules. Le reste de cette section est dédiée à `evalExpression()` et les deux fonctions `evalTerm()` et `evalFactor()`. Le code est un peu compliqué, mais il est introduit ici pour compléter l'application. Etant donné que le code n'est pas lié à la programmation d'interfaces graphiques utilisateurs, vous pouvez tranquillement l'ignorer et continuer à lire le Chapitre 5.

La fonction `evalExpression()` retourne la valeur d'une expression dans la feuille de calcul. Une expression est définie sous la forme d'un ou plusieurs termes séparés par les opérateurs "+" ou "–".Les termes eux-mêmes sont définis comme un ou plusieurs facteurs séparés par les opérateurs "*" ou "/".En divisant les expressions en termes et les termes en facteurs, nous sommes sûrs que les opérateurs sont appliqués dans le bon ordre.

Par exemple, "2*C5+D6" est une expression avec "2*C5" comme premier terme et "D6" comme second terme. Le terme "2*C5" a "2" comme premier facteur et "C5" comme deuxième facteur, et le terme "D6" est constitué d'un seul facteur "D6".Un facteur peut être un nombre ("2"), un emplacement de cellule ("C5") ou une expression entre parenthèses, précédée facultativement d'un signe moins unaire.

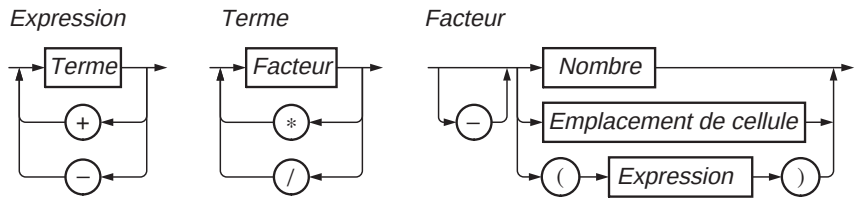


Figure 4.10
Diagramme de la syntaxe des expressions de la feuille de calcul

La syntaxe des expressions de la feuille de calcul est présentée dans la Figure 4.10. Pour chaque symbole de la grammaire (*Expression*, *Terme* et *Facteur*), il existe une fonction membre correspondante qui l'analyse et dont la structure respecte scrupuleusement cette grammaire. Les analyseurs écrits de la sorte sont appelés des analyseurs vers le bas récursifs.

Commençons par `evalExpression()`, la fonction qui analyse une *Expression :*

```
QVariant Cell::evalExpression(const QString &str, int &pos) const
{
    QVariant result = evalTerm(str, pos);
    while (str[pos] != QChar::Null) {
        QChar op = str[pos];
        if (op != '+' && op != '-')
            return result;
        ++pos;
```

```
        QVariant term = evalTerm(str, pos);
        if (result.type() == QVariant::Double
                && term.type() == QVariant::Double) {
            if (op == '+') {
                result = result.toDouble() + term.toDouble();
            } else {
            result = result.toDouble() - term.toDouble();
            }
        } else {
            result = Invalid;
        }
    }
    return result;
}
```

Nous appelons tout d'abord `evalTerm()` pour obtenir la valeur du premier terme. Si le caractère suivant est "+" ou "–", nous appelons `evalTerm()` une deuxième fois ; sinon, l'expression est constituée d'un seul terme et nous retournons sa valeur en tant que valeur de toute l'expression. Une fois que nous avons les valeurs des deux premiers termes, nous calculons le résultat de l'opération en fonction de l'opérateur. Si les deux termes ont été évalués en `double`, nous calculons le résultat comme étant `double` ; sinon nous définissons le résultat comme étant `Invalid`.

Nous continuons de cette manière jusqu'à ce qu'il n'y ait plus de termes. Ceci fonctionne correctement parce que les additions et les soustractions sont de type associatif gauche ; c'est-à-dire que "1–2–3" signifie "(1–2)–3" et non "1–(2–3)."

```
QVariant Cell::evalTerm(const QString &str, int &pos) const
{
    QVariant result = evalFactor(str, pos);
    while (str[pos] != QChar::Null) {
        QChar op = str[pos];
        if (op != '*' && op != '/')
            return result;
        ++pos;

        QVariant factor = evalFactor(str, pos);
        if (result.type() == QVariant::Double
                && factor.type() == QVariant::Double) {
            if (op == '*') {
                result = result.toDouble() * factor.toDouble();
            } else {
                if (factor.toDouble() == 0.0) {
                    result = Invalid;
            } else {
                result = result.toDouble() / factor.toDouble();
             }
          }
       } else {
```

```
            result = Invalid;
        }
    }
    return result;
}
```

La fonction `evalTerm()` ressemble beaucoup à `evalExpression()`, sauf qu'elle traite des multiplications et des divisions. La seule subtilité dans `evalTerm()`, c'est que vous devez éviter de diviser par zéro, parce que cela constitue une erreur dans certains processeurs. Bien qu'il ne soit pas recommandé de tester l'égalité de valeurs de type virgule flottante en raison des problèmes d'arrondis, vous pouvez tester sans problèmes l'égalité par rapport à 0,0 pour éviter toute division par zéro.

```
QVariant Cell::evalFactor(const QString &str, int &pos) const
{
    QVariant result;
    bool negative = false;

    if (str[pos] == '-') {
        negative = true;
        ++pos;
    }

    if (str[pos] == '(') {
        ++pos;
        result = evalExpression(str, pos);
        if (str[pos] != ')')
          result = Invalid;
         ++pos;
    } else {
        QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
        QString token;

        while (str[pos].isLetterOrNumber() || str[pos] == '.') {
            token += str[pos];
            ++pos;
        }

        if (regExp.exactMatch(token)) {
            int column = token[0].toUpper().unicode() - 'A';
            int row = token.mid(1).toInt() - 1;

            Cell *c = static_cast<Cell *>(
                            tableWidget()->item(row, column));
            if (c) {
                result = c->value();
            } else {
                result = 0.0;
            }
        } else {
            bool ok;
            result = token.toDouble(&ok);
```

```
            if (!ok)
               result = Invalid;
         }
      }

      if (negative) {
          if (result.type() == QVariant::Double) {
              result = -result.toDouble();
          } else {
             result = Invalid;
          }
      }
      return result;
   }
```

La fonction `evalFactor()` est un peu plus compliquée que `evalExpression()` et `evalTerm()`. Nous regardons d'abord si le facteur est précédé du signe négatif. Nous examinons ensuite s'il commence par une parenthèse ouverte. Si c'est le cas, nous évaluons le contenu des parenthèses comme une expression en appelant `evalExpression()`. Lorsque nous évaluons une expression entre parenthèses, `evalExpression()` appelle `evalTerm()`, qui invoque `evalFactor()`, qui appelle à nouveau `evalExpression()`. C'est là qu'intervient la récursivité dans l'analyseur.

Si le facteur n'est pas une expression imbriquée, nous extrayons le prochain jeton, qui devrait être un emplacement de cellule ou un nombre. Si le jeton correspond à `QRegExp`, nous le considérons comme une référence de cellule et nous appelons `value()` sur la cellule à l'emplacement donné. La cellule pourrait se trouver n'importe où dans la feuille de calcul et pourrait être dépendante d'autres cellules. Les dépendances ne sont pas un problème ; elles déclencheront simplement plus d'appels de `value()` et (pour les cellules "à recalculer") plus d'analyse jusqu'à ce que les valeurs des cellules dépendantes soient calculées. Si le jeton n'est pas un emplacement de cellule, nous le considérons comme un nombre.

Que se passe-t-il si la cellule A1 contient la formule "=A1" ? Ou si la cellule A1 contient "=A2" et la cellule A2 comporte "=A1" ? Même si nous n'avons pas écrit de code spécial pour détecter des dépendances circulaires, l'analyseur gère ces cas en retournant un `QVariant` invalide. Ceci fonctionne parce que nous définissons `cacheIsDirty` en `false` et `cachedValue` en `Invalid` dans `value()` avant d'appeler `evalExpression()`. Si `evalExpression()` appelle de manière récursive `value()` sur la même cellule, il renvoie immédiatement `Invalid` et toute l'expression est donc évaluée en `Invalid`.

Nous avons désormais terminé l'analyseur de formules. Il n'est pas compliqué de l'étendre pour qu'il gère des fonctions prédéfinies de la feuille de calcul, comme `sum()` et `avg()`, en développant la définition grammaticale de *Facteur*. Une autre extension facile consiste à implémenter l'opérateur "+" avec des opérandes de chaîne (comme une concaténation) ; aucun changement de grammaire n'est exigé.

5

Créer des widgets personnalisés

Au sommaire de ce chapitre

- ✔ Personnaliser des widgets Qt
- ✔ Dériver QWidget
- ✔ Intégrer des widgets personnalisés avec le Qt Designer
- ✔ Double mise en mémoire tampon

Ce chapitre vous explique comment concevoir des widgets personnalisés à l'aide de Qt. Les widgets personnalisés peuvent être créés en dérivant un widget Qt existant ou en dérivant directement `QWidget`. Nous vous présenterons les deux approches et nous verrons également comment introduire un widget personnalisé avec le *Qt Designer* de sorte qu'il puisse être utilisé comme n'importe quel widget Qt intégré. Nous terminerons ce chapitre en vous parlant d'un widget personnalisé qui emploie la double mise en mémoire tampon, une technique puissante pour actualiser très rapidement l'affichage.

Personnaliser des widgets Qt

Il arrive qu'il ne soit pas possible d'obtenir la personnalisation requise pour un widget Qt simplement en configurant ses propriétés dans le *Qt Designer* ou en appelant ses fonctions. Une solution simple et directe consiste à dériver la classe de widget appropriée et à l'adapter pour satisfaire vos besoins.

Figure 5.1
Le widget HexSpinBox

Dans cette section, nous développerons un pointeur toupie hexadécimal pour vous présenter son fonctionnement (voir Figure 5.1). `QSpinBox` ne prend en charge que les nombres décimaux, mais grâce à la dérivation, il est plutôt facile de lui faire accepter et afficher des valeurs hexadécimales.

```
#ifndef HEXSPINBOX_H
#define HEXSPINBOX_H

#include <QSpinBox>

class QRegExpValidator;

class HexSpinBox : public QSpinBox
{
    Q_OBJECT

public:
   HexSpinBox(QWidget *parent = 0);

protected:
   QValidator::State validate(QString &text, int &pos) const;
   int valueFromText(const QString &text) const;
   QString textFromValue(int value) const;

private:
   QRegExpValidator *validator;
};

#endif
```

`HexSpinBox` hérite la majorité de ses fonctionnalités de `QSpinBox`. Il propose un constructeur typique et réimplémente trois fonctions virtuelles de `QSpinBox.`

```
#include <QtGui>

#include "hexspinbox.h"

HexSpinBox::HexSpinBox(QWidget *parent)
```

```
    : QSpinBox(parent)
{
    setRange(0, 255);
    validator = new QRegExpValidator(QRegExp("[0-9A-Fa-f]{1,8}"), this);
}
```

Nous définissons la plage par défaut avec les valeurs 0 à 255 (0x00 à 0xFF), qui est plus appropriée pour un pointeur toupie hexadécimal que les valeurs par défaut de `QSpinBox` allant de 0 à 99.

L'utilisateur peut modifier la valeur en cours d'un pointeur toupie, soit en cliquant sur ses flèches vers le haut et le bas, soit en saisissant une valeur dans son éditeur de lignes. Dans le second cas, nous souhaitons restreindre l'entrée de l'utilisateur aux nombres hexadécimaux valides. Pour ce faire, nous employons `QRegExpValidator` qui accepte entre un et huit caractères, chacun d'eux devant appartenir à l'un des ensembles suivants, "0" à "9," "A" à "F" et "a" à "f".

```
QValidator::State HexSpinBox::validate(QString &text, int &pos) const
{
    return validator->validate(text, pos);
}
```

Cette fonction est appelée par `QSpinBox` pour vérifier que le texte saisi jusqu'à présent est valide. Il y a trois possibilités : `Invalid` (le texte ne correspond pas à l'expression régulière), `Intermediate` (le texte est une partie plausible d'une valeur valide) et `Acceptable` (le texte est valide). `QRegExpValidator` possède une fonction `validate()` appropriée, nous retournons donc simplement le résultat de son appel. En théorie, nous devrions renvoyer `Invalid` ou `Intermediate` pour les valeurs qui se situent en dehors de la plage du pointeur toupie, mais `QSpinBox` est assez intelligent pour détecter cette condition sans aucune aide.

```
QString HexSpinBox::textFromValue(int value) const
{
    return QString::number(value, 16).toUpper();
}
```

La fonction `textFromValue()` convertit une valeur entière en chaîne. `QSpinBox` l'appelle pour mettre à jour la partie "éditeur" du pointeur toupie quand l'utilisateur appuie sur les flèches haut et bas du pointeur. Nous utilisons la fonction statique `QString::number()` avec un second argument de 16 pour convertir la valeur en hexadécimal minuscule et nous appelons `QString::toUpper()` sur le résultat pour le passer en majuscule.

```
int HexSpinBox::valueFromText(const QString &text) const
{
    bool ok;
    return text.toInt(&ok, 16);
}
```

La fonction `valueFromText()` effectue une conversion inverse, d'une chaîne en une valeur entière. Elle est appelée par `QSpinBox` quand l'utilisateur saisit une valeur dans la zone de l'éditeur du pointeur toupie et appuie sur Entrée. Nous exécutons la fonction `QString::toInt()`

pour essayer de convertir le texte en cours en une valeur entière, toujours en base 16. Si la chaîne n'est pas au format hexadécimal, `ok` est défini en `false` et `toInt()` retourne 0. Ici, nous ne sommes pas obligés d'envisager cette possibilité, parce que le validateur n'accepte que la saisie de chaînes hexadécimales valides. Au lieu de transmettre l'adresse d'une variable sans intérêt (`ok`), nous pourrions transmettre un pointeur nul comme premier argument de `toInt()`.

Nous avons terminé le pointeur toupie hexadécimal. La personnalisation d'autres widgets Qt suit le même processus : choisir un widget Qt adapté, le dériver et réimplémenter certaines fonctions virtuelles pour modifier son comportement.

Dériver QWidget

De nombreux widgets personnalisés sont simplement obtenus à partir d'une combinaison de widgets existants, que ce soit des widgets Qt intégrés ou d'autres widgets personnalisés comme `HexSpinBox`. Les widgets personnalisés ainsi conçus peuvent généralement être développés dans le *Qt Designer* :

- créez un nouveau formulaire à l'aide du modèle "Widget" ;
- ajoutez les widgets nécessaires au formulaire, puis disposez-les ;
- établissez les connexions entre les signaux et les slots.
- Si vous avez besoin d'un comportement pour lequel de simples signaux et slots sont insuffisants, écrivez le code nécessaire dans une classe qui hérite de `QWidget` et de celle générée par `uic`.

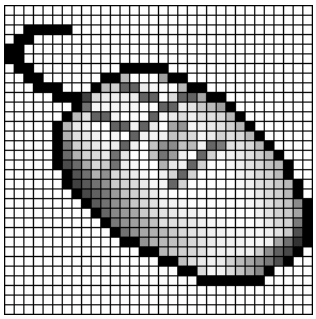
Il est évident que combiner des widgets existants peut se faire entièrement dans du code. Quelle que soit l'approche choisie, la classe en résultant hérite directement de `QWidget`.

Si le widget ne possède aucun signal ni slot et qu'il ne réimplémente pas de fonction virtuelle, il est même possible de concevoir le widget simplement en combinant des widgets existants sans sous-classe. C'est la technique que nous avons employée dans le Chapitre 1 pour créer l'application Age, avec `QWidget`, `QSpinBox` et `QSlider`. Pourtant nous aurions pu tout aussi facilement dériver `QWidget` et créer `QSpinBox` et `QSlider` dans le constructeur de la sous-classe.

Lorsqu'aucun des widgets Qt ne convient à une tâche particulière et lorsqu'il n'existe aucun moyen de combiner ou d'adapter des widgets existants pour obtenir le résultat souhaité, nous pouvons toujours créer le widget que nous désirons. Pour ce faire, nous devons dériver `QWidget` et réimplémenter quelques gestionnaires d'événements pour dessiner le widget et répondre aux clics de souris. Cette approche nous autorise une liberté totale quant à la définition et au contrôle de l'apparence et du comportement de notre widget. Les widgets intégrés de Qt, comme `QLabel`, `QPushButton` et `QTableWidget`, sont implémentés de cette manière. S'ils n'existaient pas dans Qt, il serait encore possible de les créer nous-mêmes en utilisant les fonctions publiques fournies par `QWidget` de façon totalement indépendante de la plate-forme.

Pour vous montrer comment écrire un widget personnalisé en se basant sur cette technique, nous allons créer le widget `IconEditor` illustré en Figure 5.2. `IconEditor` est un widget qui pourrait être utilisé dans un programme d'éditeur d'icônes.

Figure 5.2
Le widget IconEditor



Commençons par analyser le fichier d'en-tête.

```
#ifndef ICONEDITOR_H
#define ICONEDITOR_H

#include <QColor>
#include <QImage>
#include <QWidget>

class IconEditor : public QWidget
{
   Q_OBJECT
   Q_PROPERTY(QColor penColor READ penColor WRITE setPenColor)
   Q_PROPERTY(QImage iconImage READ iconImage WRITE setIconImage)
   Q_PROPERTY(int zoomFactor READ zoomFactor WRITE setZoomFactor)

public:
   IconEditor(QWidget *parent = 0);

   void setPenColor(const QColor &newColor);
   QColor penColor() const { return curColor; }

   void setZoomFactor(int newZoom);
   int zoomFactor() const { return zoom; }
   void setIconImage(const QImage &newImage);
   QImage iconImage() const { return image; }
   QSize sizeHint() const;
```

La classe `IconEditor` utilise la macro `Q_PROPERTY()` pour déclarer trois propriétés personnalisées : `penColor`, `iconImage` et `zoomFactor`. Chaque propriété a un type de données, une fonction de "lecture" et une fonction facultative "d'écriture".Par exemple, la propriété `penColor` est de type `QColor` et peut être lue et écrite grâce aux fonctions `penColor()` et `setPenColor()`.

Quand nous utilisons le widget dans le *Qt Designer*, les propriétés personnalisées apparaissent dans l'éditeur de propriétés du *Qt Designer* sous les propriétés héritées de `QWidget`. Ces propriétés peuvent être de n'importe quel type pris en charge par `QVariant`. La macro `Q_OBJECT` est nécessaire pour les classes qui définissent des propriétés.

```
protected:
   void mousePressEvent(QMouseEvent *event);
   void mouseMoveEvent(QMouseEvent *event);
   void paintEvent(QPaintEvent *event);

private:
   void setImagePixel(const QPoint &pos, bool opaque);
   QRect pixelRect(int i, int j) const;

   QColor curColor;
   QImage image;
   int zoom;
};

#endif
```

`IconEditor` réimplémente trois fonctions protégées de `QWidget` et possède quelques fonctions et variables privées. Les trois variables privées contiennent les valeurs des trois propriétés.

Le fichier d'implémentation commence par le constructeur de `IconEditor` :

```
#include <QtGui>

#include "iconeditor.h"

IconEditor::IconEditor(QWidget *parent)
   : QWidget(parent)
{
   setAttribute(Qt::WA_StaticContents);
   setSizePolicy(QSizePolicy::Minimum, QSizePolicy::Minimum);

   curColor = Qt::black;
   zoom = 8;

   image = QImage(16, 16, QImage::Format_ARGB32);
   image.fill(qRgba(0, 0, 0, 0));
}
```

Le constructeur présente certains aspects subtils, tels que l'attribut `Qt::WA_StaticContents` et l'appel de `setSizePolicy()`. Nous y reviendrons dans un instant.

La couleur du crayon est définie en noir. Le facteur de zoom est de 8, ce qui signifie que chaque pixel de l'icône sera affiché sous forme d'un carré de 8×8.

Les données de l'icône sont stockées dans la variable membre `image` et sont disponibles par le biais des fonctions `setIconImage()` et `iconImage()`. Un programme d'éditeur d'icônes appellerait normalement `setIconImage()` quand l'utilisateur ouvre un fichier d'icône et

`iconImage()` pour récupérer l'icône quand l'utilisateur veut la sauvegarder. La variable `image` est de type `QImage`. Nous l'initialisons à 16 × 16 pixels et au format ARGB 32 bits, un format qui prend en charge la semi-transparence. Nous effaçons les données de l'image en la remplissant avec une couleur transparente.

La classe `QImage` stocke une image indépendamment du matériel. Elle peut être définie avec une qualité de 1, 8 ou 32 bits. Une image avec une qualité de 32 bits utilise 8 bits pour chaque composante rouge, vert et bleu d'un pixel. Les 8 bits restants stockent le canal alpha du pixel (opacité). Par exemple, les composantes rouge, vert, bleu et alpha d'une couleur rouge pure présentent les valeurs 255, 0, 0 et 255. Dans Qt, cette couleur peut être spécifiée comme telle :

```
QRgb red = qRgba(255, 0, 0, 255);
```

ou, étant donné que la couleur est opaque, comme

```
QRgb red = qRgb(255, 0, 0);
```

`QRgb` est simplement le typedef d'un type `unsigned int`, et `qRgb()` et `qRgba()` sont des fonctions en ligne qui combinent leurs arguments en une valeur entière 32 bits. Il est aussi possible d'écrire

```
QRgb red = 0xFFFF0000;
```

où le premier `FF` correspond au canal alpha et le second `FF` à la composante rouge. Dans le constructeur de `IconEditor`, nous remplissons `QImage` avec une couleur transparente en utilisant 0 comme canal alpha.

Qt propose deux types permettant de stocker les couleurs : `QRgb` et `QColor`. Alors que `QRgb` est un simple typedef employé dans `QImage` pour stocker les données 32 bits du pixel, `QColor` est une classe dotée de nombreuses fonctions pratiques qui est souvent utilisée dans Qt pour stocker des couleurs. Dans le widget `IconEditor`, nous employons uniquement `QRgb` lorsque nous travaillons avec `QImage` ; nous utilisons `QColor` pour tout le reste, notamment la propriété `penColor`.

```
QSize IconEditor::sizeHint() const
{
   QSize size = zoom * image.size();
   if (zoom >= 3)
      size += QSize(1, 1);
   return size;
}
```

La fonction `sizeHint()` est réimplémentée dans `QWidget` et retourne la taille idéale d'un widget. Dans ce cas, nous recevons la taille de l'image multipliée par le facteur de zoom, avec un pixel supplémentaire dans chaque direction pour s'adapter à une grille si le facteur de zoom est de 3 ou plus. (Nous n'affichons pas de grille si le facteur de zoom est de 2 ou 1, parce qu'elle ne laisserait presque pas de place pour les pixels de l'icône.)

La taille requise d'un widget est utile dans la plupart des cas lorsqu'elle est associée aux dispositions. Les gestionnaires de disposition de Qt essaient au maximum de respecter cette taille quand ils disposent les widgets enfants d'un formulaire. Pour que `IconEditor` se comporte correctement, il doit signaler une taille requise crédible.

En plus de cette taille requise, la taille des widgets suit une stratégie qui indique au système de disposition s'ils peuvent être étirés ou rétrécis. En appelant `setSizePolicy()` dans le constructeur avec les stratégies horizontale et verticale `QSizePolicy::Minimum`, tout gestionnaire de disposition responsable de ce widget sait que la taille requise de ce dernier correspond vraiment à sa taille minimale. En d'autres termes, le widget peut être étiré si nécessaire, mais ne doit jamais être rétréci à une taille inférieure à la taille requise. Vous pouvez annuler ce comportement dans le *Qt Designer* en configurant la propriété `sizePolicy` du widget. La signification des diverses stratégies liées à la taille est expliquée au Chapitre 6.

```
void IconEditor::setPenColor(const QColor &newColor)
{
   curColor = newColor;
}
```

La fonction `setPenColor()` définit la couleur du crayon. La couleur sera utilisée pour les pixels que vous dessinerez.

```
void IconEditor::setIconImage(const QImage &newImage)
{
   if (newImage != image) {
      image = newImage.convertToFormat(QImage::Format_ARGB32);
      update();
      updateGeometry();
   }
}
```

La fonction `setIconImage()` détermine l'image à modifier. Nous invoquons `convertToFormat()` pour obtenir une image 32 bits avec une mémoire tampon alpha si elle n'est pas dans ce format. Ailleurs dans le code, nous supposerons que les données de l'image sont stockées sous forme de valeurs ARGB 32 bits.

Après avoir configuré la variable `image`, nous appelons `QWidget::update()` pour forcer le rafraîchissement de l'affichage du widget avec la nouvelle image. Nous invoquons ensuite `QWidget::updateGeometry()` pour informer toute disposition qui contient le widget que la taille requise du widget a changé. La disposition s'adaptera automatiquement à cette nouvelle taille.

```
void IconEditor::setZoomFactor(int newZoom)
{
   if (newZoom < 1)
       newZoom = 1;

   if (newZoom != zoom) {
       zoom = newZoom;
```

```
            update();
            updateGeometry();
        }
    }
```

La fonction `setZoomFactor()` définit le facteur de zoom de l'image. Pour éviter une division par zéro, nous corrigeons toute valeur inférieure à 1. A nouveau, nous appelons `update()` et `updateGeometry()` pour actualiser l'affichage du widget et pour informer tout gestionnaire de disposition de la modification de la taille requise.

Les fonctions `penColor()`, `iconImage()` et `zoomFactor()` sont implémentées en tant que fonctions en ligne dans le fichier d'en-tête.

Nous allons maintenant passer en revue le code de la fonction `paintEvent()`. Cette fonction est la fonction la plus importante de `IconEditor`. Elle est invoquée dès que le widget a besoin d'être redessiné. L'implémentation par défaut dans `QWidget` n'a aucune conséquence, le widget reste vide.

Tout comme `closeEvent()`, que nous avons rencontré dans le Chapitre 3, `paintEvent()` est un gestionnaire d'événements. Qt propose de nombreux autres gestionnaires d'événements, chacun d'eux correspondant à un type différent d'événement. Le Chapitre 7 aborde en détail le traitement des événements.

Il existe beaucoup de situations où un événement `paint` est déclenché et où `paintEvent()` est appelé :

- Quand un widget est affiché pour la première fois, le système génère automatiquement un événement `paint` pour obliger le widget à se dessiner lui-même.
- Quand un widget est redimensionné, le système déclenche un événement `paint`.
- Si le widget est masqué par une autre fenêtre, puis affiché à nouveau, un événement `paint` est déclenché pour la zone qui était masquée (à moins que le système de fenêtrage ait stocké la zone).

Nous avons aussi la possibilité de forcer un événement `paint` en appelant `QWidget::update()` ou `QWidget::repaint()`. La différence entre ces deux fonctions est que `repaint()` impose un rafraîchissement immédiat de l'affichage, alors que `update()` planifie simplement un événement `paint` pour le prochain traitement d'événements de Qt. (Ces deux fonctions ne font rien si le widget n'est pas visible à l'écran.) Si `update()` est invoqué plusieurs fois, Qt compresse les événements `paint` consécutifs en un seul événement `paint` pour éviter le phénomène du scintillement. Dans `IconEditor`, nous utilisons toujours `update()`.

Voici le code :

```
void IconEditor::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);

    if (zoom >= 3) {
        painter.setPen(palette().foreground().color());
        for (int i = 0; i <= image.width(); ++i)
```

```
            painter.drawLine(zoom * i, 0,
                             zoom * i, zoom * image.height());
        for (int j = 0; j <= image.height(); ++j)
            painter.drawLine(0, zoom * j,
                             zoom * image.width(), zoom * j);
    }

    for (int i = 0; i < image.width(); ++i) {
        for (int j = 0; j < image.height(); ++j) {
            QRect rect = pixelRect(i, j);
            if (!event->region().intersect(rect).isEmpty()) {
                QColor color = QColor::fromRgba(image.pixel(i, j));
                painter.fillRect(rect, color);
            }
        }
    }
}
```

Nous commençons par construire un objet `QPainter` sur le widget. Si le facteur de zoom est de 3 ou plus, nous dessinons des lignes horizontales et verticales qui forment une grille à l'aide de la fonction `QPainter::drawLine()`.

Un appel de `QPainter::drawLine()` présente la syntaxe suivante :

```
painter.drawLine(x1, y1, x2, y2);
```

où `(x1, y1)` est la position d'une extrémité de la ligne et `(x2, y2)` la position de l'autre extrémité. Il existe également une version surchargée de la fonction qui reçoit deux `QPoint` au lieu de quatre `int`.

Le pixel en haut à gauche d'un widget Qt se situe à la position (0, 0), et le pixel en bas à droite se trouve à `(width() - 1, height() - 1)`. Cela ressemble au système traditionnel de coordonnées cartésiennes, mais à l'envers. Nous avons la possibilité de modifier le système de coordonnées de `QPainter` grâce aux transformations, comme la translation, la mise à l'échelle, la rotation et le glissement. Ces notions sont abordées au Chapitre 8 (Graphiques 2D et 3D).

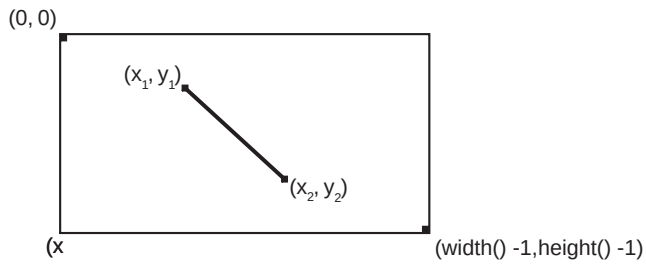


Figure 5.3
Tracer une ligne avec QPainter

Avant d'appeler `drawLine()` sur `QPainter`, nous définissons la couleur de la ligne au moyen de `setPen()`. Nous pourrions coder une couleur, comme noir ou gris, mais il est plus judicieux d'utiliser la palette du widget.

Chaque widget est doté d'une palette qui spécifie quelles couleurs doivent être utilisées selon les situations. Par exemple, il existe une entrée dans la palette pour la couleur d'arrière-plan des widgets (généralement gris clair) et une autre pour la couleur du texte sur ce fond (habituellement noir). Par défaut, la palette d'un widget adopte le modèle de couleur du système de fenêtrage. En utilisant des couleurs de la palette, nous sommes sûrs que `IconEditor` respecte les préférences de l'utilisateur.

La palette d'un widget consiste en trois groupes de couleurs : `active`, `inactive` et `disabled`. Vous choisirez le groupe de couleurs en fonction de l'état courant du widget :

- Le groupe `Active` est employé pour des widgets situés dans la fenêtre actuellement active.
- Le groupe `Inactive` est utilisé pour les widgets des autres fenêtres.
- Le groupe `Disabled` est utilisé pour les widgets désactivés dans n'importe quelle fenêtre.

La fonction `QWidget::palette()` retourne la palette du widget sous forme d'objet `QPalette`. Les groupes de couleurs sont spécifiés comme des énumérations de type `QPalette::ColorGroup`.

Lorsque nous avons besoin d'un pinceau ou d'une couleur appropriée pour dessiner, la bonne approche consiste à utiliser la palette courante, obtenue à partir de `QWidget::palette()`, et le rôle requis, par exemple, `QPalette::foreground()`. Chaque fonction de rôle retourne un pinceau, qui correspond normalement à ce que nous souhaitons, mais si nous n'avons besoin que de la couleur, nous pouvons l'extraire du pinceau, comme nous avons fait dans `paintEvent()`. Par défaut, les pinceaux retournés sont adaptés à l'état du widget, nous ne sommes donc pas forcés de spécifier un groupe de couleurs.

La fonction `paintEvent()` termine en dessinant l'image elle-même. L'appel de `IconEditor::pixelRect()` retourne un `QRect` qui définit la région à redessiner. Pour une question d'optimisation simple, nous ne redessinons pas les pixels qui se trouvent en dehors de cette région.

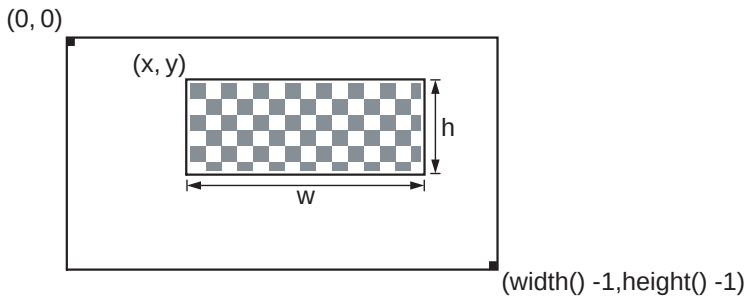


Figure 5.4
Dessiner un rectangle avec QPainter

Nous invoquons `QPainter::fillRect()` pour dessiner un pixel sur lequel un zoom a été effectué. `QPainter::fillRect()` reçoit un `QRect` et un `QBrush`. En transmettant `QColor` comme pinceau, nous obtenons un modèle de remplissage correct.

```
QRect IconEditor::pixelRect(int i, int j) const
{
   if (zoom >= 3) {
      return QRect(zoom * i + 1, zoom * j + 1, zoom - 1, zoom - 1);
   } else {
      return QRect(zoom * i, zoom * j, zoom, zoom);
   }
}
```

La fonction `pixelRect()` retourne un `QRect` adapté à `QPainter::fillRect()`. Les paramètres `i` et `j` sont les coordonnées du pixel dans `QImage` – pas dans le widget. Si le facteur de zoom est de 1, les deux systèmes de coordonnées coïncident parfaitement.

Le constructeur de `QRect` suit la syntaxe `QRect(x, y, width, height)`, où `(x, y)` est la position du coin supérieur gauche du rectangle et `width _ height` correspond à la taille du rectangle. Si le facteur de zoom est de 3 ou plus, nous réduisons la taille du rectangle d'un pixel horizontalement et verticalement, de sorte que le remplissage ne déborde pas sur les lignes de la grille.

```
void IconEditor::mousePressEvent(QMouseEvent *event)
{
   if (event->button() == Qt::LeftButton) {
       setImagePixel(event->pos(), true);
   } else if (event->button() == Qt::RightButton) {
       setImagePixel(event->pos(), false);
   }
}
```

Quand l'utilisateur appuie sur un bouton de la souris, le système déclenche un événement "bouton souris enfoncé". En réimplémentant `QWidget::mousePressEvent()`, nous avons la possibilité de répondre à cet événement et de définir ou effacer le pixel de l'image sous le pointeur de la souris.

Si l'utilisateur a appuyé sur le bouton gauche de la souris, nous appelons la fonction privée `setImagePixel()` avec `true` comme second argument, lui demandant de définir le pixel dans la couleur actuelle du crayon. Si l'utilisateur a appuyé sur le bouton droit de la souris, nous invoquons aussi `setImagePixel()`, mais nous transmettons `false` pour effacer le pixel.

```
void IconEditor::mouseMoveEvent(QMouseEvent *event)
{
   if (event->buttons() & Qt::LeftButton) {
       setImagePixel(event->pos(), true);
   } else if (event->buttons() & Qt::RightButton) {
       setImagePixel(event->pos(), false);
   }
}
```

`mouseMoveEvent()` gère les événements "déplacement de souris".Par défaut, ces événements ne sont déclenchés que lorsque l'utilisateur enfonce un bouton. Il est possible de changer ce comportement en appelant `QWidget::setMouseTracking()`, mais nous n'avons pas besoin d'agir de la sorte dans cet exemple.

Tout comme le fait d'appuyer sur les boutons droit ou gauche de la souris configure ou efface un pixel, garder ce bouton enfoncé et se placer sur un pixel suffit aussi à définir ou supprimer un pixel. Vu qu'il est possible de maintenir enfoncé plus d'un bouton à la fois, la valeur retournée par `QMouseEvent::buttons()` est un opérateur `OR` bit à bit des boutons de la souris. Nous testons si un certain bouton est enfoncé à l'aide de l'opérateur `&`, et si c'est le cas, nous invoquons `setImagePixel()`.

```
void IconEditor::setImagePixel(const QPoint &pos, bool opaque)
{
   int i = pos.x() / zoom;
   int j = pos.y() / zoom;

   if (image.rect().contains(i, j)) {
      if (opaque) {
          image.setPixel(i, j, penColor().rgba());
      } else {
          image.setPixel(i, j, qRgba(0, 0, 0, 0));
      }

      update(pixelRect(i, j));
   }
}
```

La fonction `setImagePixel()` est appelée depuis `mousePressEvent()` et `mouseMoveEvent()` pour définir ou effacer un pixel. Le paramètre `pos` correspond à la position de la souris dans le widget.

La première étape consiste à convertir la position de la souris dans les coordonnées du widget vers les coordonnées de l'image. Pour ce faire, les composants `x()` et `y()` de la position de la souris sont divisés par le facteur de zoom. Puis nous vérifions si le point se trouve dans une plage correcte. Ce contrôle s'effectue facilement en utilisant `QImage::rect()` et `QRect::contains()` ; vous vérifiez ainsi que `i` se situe entre 0 et `image.width() - 1` et que `j` est entre 0 et `image.height() - 1`.

Selon le paramètre opaque, nous définissons ou nous effaçons le pixel dans l'image. Effacer un pixel consiste à le rendre transparent. Nous devons convertir le crayon `QColor` en une valeur ARGB 32 bits pour l'appel de `QImage::setPixel()`. Nous terminons en appelant `update()` avec un `QRect` de la zone qui doit être redessinée.

Maintenant que nous avons analysé les fonctions membres, nous allons retourner l'attribut `Qt::WA_StaticContents` que nous avons utilisé dans le constructeur. Cet attribut informe Qt que le contenu du widget ne change pas quand le widget est redimensionné et que le contenu reste ancré dans le coin supérieur gauche du widget. Qt se sert de ces informations pour éviter

tout retraçage inutile des zones qui sont déjà affichées au moment du redimensionnement du widget.

Normalement, quand un widget est redimensionné, Qt déclenche un événement `paint` pour toute la zone visible du widget (voir Figure 5.5). Mais si le widget est créé avec l'attribut `Qt::WA_StaticContents`, la région de l'événement `paint` se limite aux pixels qui n'étaient pas encore affichés auparavant. Ceci implique que si le widget est redimensionné dans une taille plus petite, aucun événement `paint` ne sera déclenché.

Figure 5.5
Redimensionner un widget Qt::WA_StaticContents

Le widget `IconEditor` est maintenant terminé. Grâce aux informations et aux exemples des chapitres précédents, nous pourrions écrire un code qui utilise `IconEditor` comme une véritable fenêtre, comme un widget central dans `QMainWindow`, comme un widget enfant dans une disposition ou comme un widget enfant dans `QScrollArea`. Dans la prochaine section, nous verrons comment l'intégrer avec le *Qt Designer.*

Intégrer des widgets personnalisés avec le Qt Designer

Avant de pouvoir utiliser des widgets personnalisés dans le *Qt Designer*, celui-ci doit en avoir connaissance. Il existe deux techniques : la "promotion" et le plug-in.

L'approche de la promotion est la plus rapide et la plus simple. Elle consiste à choisir un widget Qt intégré qui possède une API similaire à celle que nous voulons pour notre widget personnalisé, puis à saisir quelques informations à propos de ce widget personnalisé dans une boîte de dialogue du *Qt Designer.* Le widget peut ensuite être exploité dans des formulaires développés avec le *Qt Designer,* même s'il sera représenté par le widget Qt intégré associé lors de l'édition ou de la prévisualisation du formulaire.

Voici comment insérer un widget `HexSpinBox` dans un formulaire avec cette approche :

1. Créez un `QSpinBox` en le faisant glisser depuis la boîte des widgets du *Qt Designer* vers le formulaire.
2. Cliquez du bouton droit sur le pointeur toupie et sélectionnez Promote to Custom Widget dans le menu contextuel.
3. Complétez la boîte de dialogue qui s'ouvre avec "HexSpinBox" comme nom de classe et "hexspinbox.h" comme fichier d'en-tête (voir Figure 5.6).

Voilà ! Le code généré par `uic` contiendra `hexspinbox.h` au lieu de `<QSpinBox>` et instanciera un `HexSpinBox`. Dans le *Qt Designer*, le widget `HexSpinBox` sera représenté par un `QSpinBox`, ce qui nous permet de définir toutes les propriétés d'un `QSpinBox` (par exemple, la plage et la valeur actuelle).

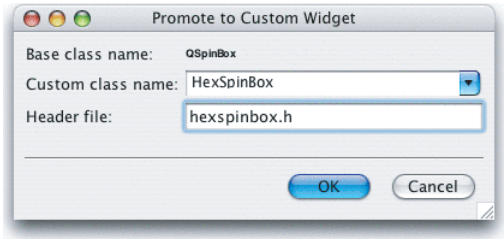


Figure 5.6
La boîte de dialogue du widget personnalisé dans le Qt Designer

Les inconvénients de l'approche de la promotion sont que les propriétés spécifiques au widget personnalisé ne sont pas accessibles dans le *Qt Designer* et que le widget n'est pas affiché en tant que tel. Ces deux problèmes peuvent être résolus en utilisant l'approche du plug-in.

L'approche du plug-in nécessite la création d'une bibliothèque de plug-in que le *Qt Designer* peut charger à l'exécution et utiliser pour créer des instances du widget. Le véritable widget est ensuite employé par le *Qt Designer* pendant la modification du formulaire et la prévisualisation, et grâce au système de méta-objets de Qt, le *Qt Designer* peut obtenir dynamiquement la liste de ses propriétés. Pour voir comment cela fonctionne, nous intégrerons le widget `IconEditor` de la section précédente comme plug-in.

Nous devons d'abord dériver `QDesignerCustomWidgetInterface` et réimplémenter certaines fonctions virtuelles. Nous supposerons que le code source du plug-in se situe dans un répertoire appelé `iconeditorplugin` et que le code source de `IconEditor` se trouve dans un répertoire parallèle nommé `iconeditor`.

Voici la définition de classe :

```
#include <QDesignerCustomWidgetInterface>

class IconEditorPlugin : public QObject,
                         public QDesignerCustomWidgetInterface
{
   Q_OBJECT
   Q_INTERFACES(QDesignerCustomWidgetInterface)

public:
   IconEditorPlugin(QObject *parent = 0);

   QString name() const;
   QString includeFile() const;
   QString group() const;
   QIcon icon() const;
   QString toolTip() const;
```

```
    QString whatsThis() const;
    bool isContainer() const;
    QWidget *createWidget(QWidget *parent);
};
```

La sous-classe `IconEditorPlugin` est une classe spécialisée qui encapsule le widget `IconEditor`. Elle hérite de `QObject` et de `QDesignerCustomWidgetIterface` et se sert de la macro `Q_INTERFACES()` pour signaler à `moc` que la seconde classe de base est une interface de plug-in. Les fonctions sont utilisées par le *Qt Designer* pour créer des instances de la classe et obtenir des informations à son sujet.

```
IconEditorPlugin::IconEditorPlugin(QObject *parent)
    : QObject(parent)
{
}
```

Le constructeur est très simple.

```
QString IconEditorPlugin::name() const
{
    return "IconEditor";
}
```

La fonction `name()` retourne le nom du widget fourni par le plug-in.

```
QString IconEditorPlugin::includeFile() const
{
    return "iconeditor.h";
}
```

La fonction `includeFile()` retourne le nom du fichier d'en-tête pour le widget spécifié encapsulé par le plug-in. Le fichier d'en-tête se trouve dans le code généré par l'outil `uic`.

```
QString IconEditorPlugin::group() const
{
    return tr("Image Manipulation Widgets");
}
```

La fonction `group()` retourne le nom du groupe de widgets auquel doit appartenir ce widget personnalisé. Si le nom n'est pas encore utilisé, le *Qt Designer* créera un nouveau groupe pour le widget.

```
QIcon IconEditorPlugin::icon() const
{
    return QIcon(":/images/iconeditor.png");
}
```

La fonction `icon()` renvoie l'icône à utiliser pour représenter le widget personnalisé dans la boîte des widgets du *Qt Designer*. Dans notre cas, nous supposons que `IconEditorPlugin` possède un fichier de ressources Qt associé avec une entrée adaptée pour l'image de l'éditeur d'icônes.

```
QString IconEditorPlugin::toolTip() const
{
   return tr("An icon editor widget");
}
```

La fonction `toolTip()` renvoie l'infobulle à afficher quand la souris se positionne sur le widget personnalisé dans la boîte des widgets du *Qt Designer*.

```
QString IconEditorPlugin::whatsThis() const
{
   return tr("This widget is presented in Chapter 5 of <i>C++ GUI "
             "Programming with Qt 4</i> as an example of a custom Qt "
             "widget.");
}
```

La fonction `whatsThis()` retourne le texte "What's this ?" que le *Qt Designer* doit afficher.

```
bool IconEditorPlugin::isContainer() const
{
   return false;
}
```

La fonction `isContainer()` retourne `true` si le widget peut contenir d'autres widgets ; sinon elle retourne `false`. Par exemple, `QFrame` est un widget qui peut comporter d'autres widgets. En général, tout widget Qt peut renfermer d'autres widgets, mais le *Qt Designer* ne l'autorise pas quand `isContainer()` renvoie `false`.

```
QWidget *IconEditorPlugin::createWidget(QWidget *parent)
{
   return new IconEditor(parent);
}
```

La fonction `create()` est invoquée par le *Qt Designer* pour créer une instance d'une classe de widget avec le parent donné.

```
Q_EXPORT_PLUGIN2(iconeditorplugin, IconEditorPlugin)
```

A la fin du fichier source qui implémente la classe de plug-in, nous devons utiliser la macro `Q_EXPORT_PLUGIN2()` pour que le *Qt Designer* puisse avoir accès au plug-in. Le premier argument est le nom que nous souhaitons attribuer au plug-in ; le second argument est le nom de la classe qui l'implémente.

Voici le code d'un fichier `.pro` permettant de générer le plug-in :

```
TEMPLATE     = lib
CONFIG      += designer plugin release
HEADERS      = ../iconeditor/iconeditor.h \
               iconeditorplugin.h
SOURCES      = ../iconeditor/iconeditor.cpp \
               iconeditorplugin.cpp
RESOURCES    = iconeditorplugin.qrc
```

```
DESTDIR       = $(QTDIR)/plugins/designer
```

Le fichier `.pro` suppose que la variable d'environnement `QTDIR` contient le répertoire où Qt est installé. Quand vous tapez `make` ou `nmake` pour générer le plug-in, il s'installera automatiquement dans le répertoire `plugins` du *Qt Designer*. Une fois le plug-in généré, le widget `IconEditor` peut être utilisé dans le *Qt Designer* de la même manière que n'importe quel autre widget intégré de Qt.

Si vous voulez intégrer plusieurs widgets personnalisés avec le *Qt Designer*, vous avez la possibilité soit de créer un plug-in pour chacun d'eux, soit de les combiner dans un seul plug-in en dérivant `QDesignerCustomWidgetCollectionInterface`.

Double mise en mémoire tampon

La double mise en mémoire tampon est une technique de programmation GUI qui consiste à afficher un widget dans un pixmap hors champ puis à copier le pixmap à l'écran. Avec les versions antérieures de Qt, cette technique était fréquemment utilisée pour éliminer le phénomène du scintillement et pour offrir une interface utilisateur plus confortable.

Dans Qt 4, `QWidget` gère ce phénomène automatiquement, nous sommes donc rarement obligés de nous soucier du scintillement des widgets. La double mise en mémoire tampon explicite reste tout de même avantageuse si le rendu du widget est complexe et doit être réalisé de façon répétitive. Nous pouvons alors stocker un pixmap de façon permanente avec le widget, toujours prêt pour le prochain événement `paint`, et copier le pixmap dans le widget dès que nous détectons cet événement `paint`. Il se révèle particulièrement utile si nous souhaitons effectuer de légères modifications, comme dessiner un rectangle de sélection, sans avoir à recalculer à chaque fois le rendu complet du widget.

Nous allons clore ce chapitre en étudiant le widget personnalisé `Plotter`. Ce widget utilise la double mise en mémoire tampon et illustre également certains aspects de la programmation Qt, notamment la gestion des événements du clavier, la disposition manuelle et les systèmes de coordonnées.

Le widget `Plotter` affiche une ou plusieurs courbes spécifiées sous forme de vecteurs de coordonnées. L'utilisateur peut tracer un rectangle de sélection sur l'image et `Plotter` zoomera sur la zone délimitée par ce tracé (voir Figure 5.7). L'utilisateur dessine le rectangle en cliquant à un endroit dans le graphique, en faisant glisser la souris vers une autre position en maintenant le bouton gauche enfoncé puis en relâchant le bouton de la souris.

L'utilisateur peut zoomer de manière répétée en traçant des rectangles de sélection plusieurs fois, faire un zoom arrière grâce au bouton Zoom Out, puis zoomer à nouveau au moyen du bouton Zoom In. Les boutons Zoom In et Zoom Out apparaissent la première fois qu'ils deviennent accessibles, ils n'encombrent donc pas l'écran si l'utilisateur ne fait pas de zoom sur le graphique.

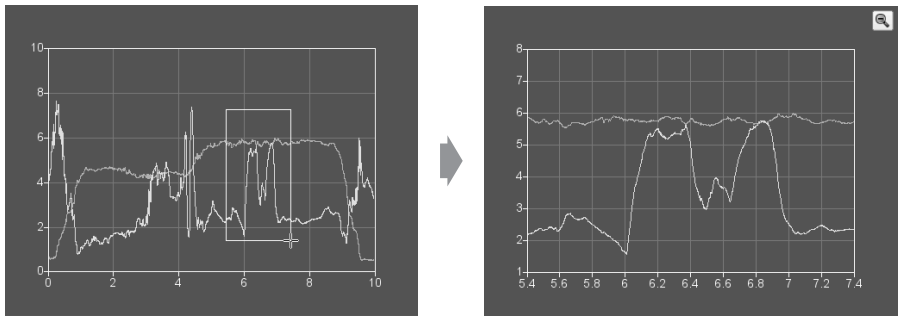


Figure 5.7
Zoomer sur le widget Plotter

Le widget `Plotter` peut enregistrer les données de nombreuses courbes. Il assure aussi la maintenance d'une pile d'objets `PlotSettings`, chacun d'eux correspondant à un niveau particulier de zoom.

Analysons désormais la classe, en commençant par `plotter.h` :

```
#ifndef PLOTTER_H
#define PLOTTER_H

#include <QMap>
#include <QPixmap>
#include <QVector>
#include <QWidget>

class QToolButton;
class PlotSettings;

class Plotter : public QWidget
{
   Q_OBJECT

public:
   Plotter(QWidget *parent = 0);

   void setPlotSettings(const PlotSettings &settings);
   void setCurveData(int id, const QVector<QPointF> &data);
   void clearCurve(int id);
   QSize minimumSizeHint() const;
   QSize sizeHint() const;

public slots:
   void zoomIn();
   void zoomOut();
```

Nous commençons par inclure les fichiers d'en-tête des classes Qt utilisées dans l'en-tête du fichier du traceur (plotter) puis nous déclarons les classes désignées par des pointeurs ou des références dans l'en-tête.

Dans la classe `Plotter`, nous fournissons trois fonctions publiques pour configurer le tracé et deux slots publics pour faire des zooms avant et arrière. Nous réimplémentons aussi `minimumSizeHint()` et `sizeHint()` dans `QWidget`. Nous enregistrons les points d'une courbe sous forme de `QVector<QPointF>`, où `QPointF` est la version virgule flottante de `QPoint`.

```
protected:
   void paintEvent(QPaintEvent *event);
   void resizeEvent(QResizeEvent *event);
   void mousePressEvent(QMouseEvent *event);
   void mouseMoveEvent(QMouseEvent *event);
   void mouseReleaseEvent(QMouseEvent *event);
   void keyPressEvent(QKeyEvent *event);
   void wheelEvent(QWheelEvent *event);
```

Dans la section protégée de la classe, nous déclarons tous les gestionnaires d'événements de `QWidget` que nous désirons réimplémenter.

```
private:
   void updateRubberBandRegion();
   void refreshPixmap();
   void drawGrid(QPainter *painter);
   void drawCurves(QPainter *painter);

   enum { Margin = 50 };

   QToolButton *zoomInButton;
   QToolButton *zoomOutButton;
   QMap<int, QVector<QPointF> > curveMap;
   QVector<PlotSettings> zoomStack;
   int curZoom;
   bool rubberBandIsShown;
   QRect rubberBandRect;
   QPixmap pixmap;
};
```

Dans la section privée de la classe, nous déclarons quelques fonctions pour dessiner le widget, une constante et quelques variables membres. La constante `Margin` sert à introduire un peu d'espace autour du graphique.

Parmi les variables membres, on compte un pixmap de type `QPixmap`. Cette variable conserve une copie du rendu de tout le widget, identique à celui affiché à l'écran. Le tracé est toujours dessiné sur ce pixmap d'abord hors champ ; puis le pixmap est copié dans le widget.

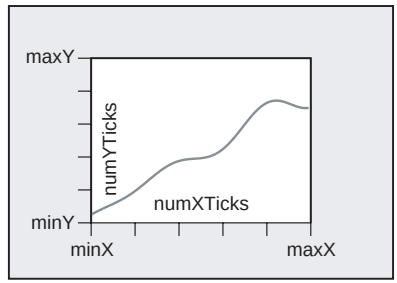
```
class PlotSettings
{
public:
   PlotSettings();

   void scroll(int dx, int dy);
   void adjust();

   double spanX() const { return maxX - minX; }
```

```
    double spanY() const { return maxY - minY; }

    double minX;
    double maxX;
    int numXTicks;
    double minY;
    double maxY;
    int numYTicks;

private:
    static void adjustAxis(double &min, double &max, int &numTicks);
};

#endif
```

La classe `PlotSettings` spécifie la plage des axes *x* et *y* et le nombre de graduations pour ces axes. La Figure 5.8 montre la correspondance entre un objet `PlotSettings` et un widget `Plotter`.

Par convention, `numXTicks` et `numYTicks` ont une unité de moins ; si `numXTicks` a la valeur 5, `Plotter` dessinera 6 graduations sur l'axe *x*. Cela simplifie les calculs par la suite.

Figure 5.8
Les variables membres de PlotSettings



Analysons à présent le fichier d'implémentation :

```
#include <QtGui>
#include <cmath>

#include "plotter.h"
```

Nous incluons les fichiers d'en-têtes prévus et nous importons tous les symboles de l'espace de noms `std` dans l'espace de noms global. Ceci nous permet d'accéder aux fonctions déclarées dans `<cmath>` sans les préfixer avec `std::` (par exemple, `floor()` au lieu de `std::floor()`).

```
Plotter::Plotter(QWidget *parent)
    : QWidget(parent)
{
    setBackgroundRole(QPalette::Dark);
    setAutoFillBackground(true);
    setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
```

```
    setFocusPolicy(Qt::StrongFocus);
    rubberBandIsShown = false;

    zoomInButton = new QToolButton(this);
    zoomInButton->setIcon(QIcon(":/images/zoomin.png"));
    zoomInButton->adjustSize();
    connect(zoomInButton, SIGNAL(clicked()), this, SLOT(zoomIn()));

    zoomOutButton = new QToolButton(this);
    zoomOutButton->setIcon(QIcon(":/images/zoomout.png"));
    zoomOutButton->adjustSize();
    connect(zoomOutButton, SIGNAL(clicked()), this, SLOT(zoomOut()));

    setPlotSettings(PlotSettings());
}
```

L'appel de `setBackgroundRole()` demande à `QWidget` d'utiliser le composant "dark" de la palette comme couleur pour effacer le widget, à la place du composant "window".Qt se voit donc attribuer une couleur par défaut qu'il peut employer pour remplir n'importe quel pixel nouvellement affiché quand le widget est redimensionné dans une taille plus grande, avant même que `paintEvent()` n'ait l'opportunité de dessiner les nouveaux pixels. Nous devons aussi invoquer `setAutoFillBackground(true)` dans le but d'activer ce mécanisme. (Par défaut, les widgets enfants héritent de l'arrière-plan de leur widget parent.)

L'appel de `setSizePolicy()` définit la stratégie de taille du widget en `QSizePolicy::Expanding` dans les deux directions. Tout gestionnaire de disposition responsable du widget sait donc que ce dernier peut être agrandi, mais peut aussi être rétréci. Ce paramètre est typique des widgets qui peuvent prendre beaucoup de place à l'écran. La valeur par défaut est `QSizePolicy::Preferred` dans les deux directions, ce qui signifie que le widget préfère être à sa taille requise, mais qu'il peut être rétréci à sa taille minimale ou agrandi à l'infini si nécessaire.

L'appel de `setFocusPolicy(Qt::StrongFocus)` permet au widget de recevoir le focus lorsque l'utilisateur clique ou appuie sur Tab. Quand le `Plotter` est actif, il recevra les événements liés aux touches du clavier enfoncées. Le widget `Plotter` réagit à quelques touches : + pour un zoom avant ; – pour un zoom arrière ; et les flèches directionnelles pour faire défiler vers la droite ou la gauche, le haut ou le bas (voir Figure 5.9).

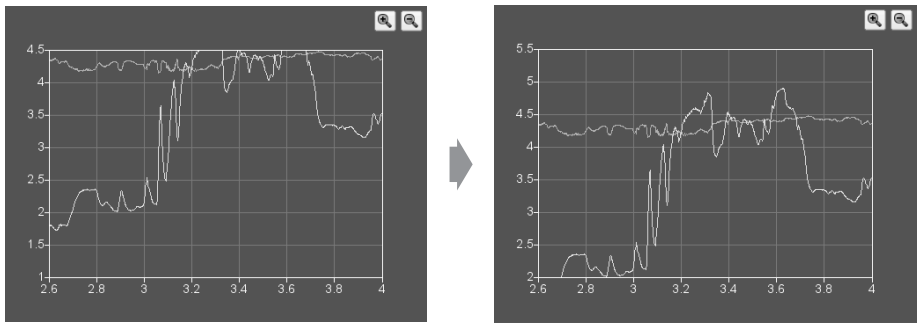


Figure 5.9
Faire défiler le widget Plotter

Toujours dans le constructeur, nous créons deux `QToolButton`, chacun avec une icône. Ces boutons permettent à l'utilisateur de faire des zooms avant et arrière. Les icônes du bouton sont stockées dans un fichier de ressources, donc toute application qui utilise le widget `Plotter` aura besoin de cette entrée dans son fichier `.pro` :

```
RESOURCES = plotter.qrc
```

Le fichier de ressources ressemble à celui que nous avons utilisé pour l'application Spreadsheet :

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
   <file>images/zoomin.png</file>
   <file>images/zoomout.png</file>
</qresource>
</RCC>
```

Les appels de `adjustSize()` sur les boutons définissent leurs tailles de sorte qu'elles correspondent à la taille requise. Les boutons ne font pas partie d'une disposition ; nous les positionnerons manuellement dans l'événement `resize` de `Plotter`. Vu que nous ne nous servons pas des dispositions, nous devons spécifier explicitement le parent des boutons en le transmettant au constructeur de `QPushButton`.

L'appel de `setPlotSettings()` à la fin termine l'initialisation.

```
void Plotter::setPlotSettings(const PlotSettings &settings)
{
   zoomStack.clear();
   zoomStack.append(settings);
   curZoom = 0;
   zoomInButton->hide();
   zoomOutButton->hide();
   refreshPixmap();
}
```

La fonction `setPlotSettings()` est employée pour spécifier le `PlotSettings` à utiliser pour afficher le tracé. Elle est appelée par le constructeur `Plotter` et peut être employée par des utilisateurs de la classe. Le traceur commence à son niveau de zoom par défaut. A chaque fois que l'utilisateur fait un zoom avant, une nouvelle instance de `PlotSettings` est créée et placée sur la pile de zoom. La pile de zoom est représentée par deux variables membres :

- `zoomStack` contient les divers paramètres de zoom sous forme de `QVector<PlotSettings>`.
- `curZoom` comporte l'index du `PlotSettings` actuel dans `zoomStack`.

Après l'appel de `setPlotSettings()`, la pile de zoom ne contient qu'une seule entrée et les boutons Zoom In et Zoom Out sont masqués. Ces boutons ne s'afficheront que lorsque nous appellerons `show()` sur eux dans les slots `zoomIn()` et `zoomOut()`. (Normalement il suffit d'invoquer `show()` sur le widget de niveau supérieur pour afficher tous les enfants.

Mais quand nous appelons explicitement `hide()` sur un widget enfant, il est masqué jusqu'à ce nous appelions à nouveau `show()` sur ce widget.)

L'appel de `refreshPixmap()` est nécessaire pour mettre à jour l'affichage. Normalement, nous invoquerions `update()`, mais dans ce cas, nous agissons légèrement différemment parce que nous voulons conserver un `QPixmap` toujours mis à jour. Après avoir régénéré le pixmap, `refreshPixmap()` appelle `update()` pour copier le pixmap dans le widget.

```
void Plotter::zoomOut()
{
   if (curZoom > 0) {
      --curZoom;
      zoomOutButton->setEnabled(curZoom > 0);
      zoomInButton->setEnabled(true);
      zoomInButton->show();
      refreshPixmap();
   }
}
```

Le slot `zoomOut()` fait un zoom arrière si vous avez déjà zoomé sur le graphique. Il décrémente le niveau actuel de zoom et active le bouton Zoom Out selon qu'il est encore possible de faire un zoom arrière ou pas. Le bouton Zoom In est activé et affiché, et l'affichage est mis à jour avec un appel de `refreshPixmap()`.

```
void Plotter::zoomIn()
{
   if (curZoom < zoomStack.count() - 1) {
      ++curZoom;
      zoomInButton->setEnabled(curZoom < zoomStack.count() - 1);
      zoomOutButton->setEnabled(true);
      zoomOutButton->show();
      refreshPixmap();
   }
}
```

Si l'utilisateur a fait un zoom avant puis un zoom arrière, le `PlotSettings` du prochain niveau de zoom sera dans la pile de zoom et nous pourrons zoomer. (Sinon, il est toujours possible de faire un zoom avant avec un rectangle de sélection.)

Le slot incrémente `curZoom` pour descendre d'un niveau dans la pile de zoom, active ou désactive le bouton Zoom In selon qu'il est possible de faire encore un zoom avant ou non, et active et affiche le bouton Zoom Out. A nouveau, nous appelons `refreshPixmap()` pour que le traceur utilise les derniers paramètres du zoom.

```
void Plotter::setCurveData(int id, const QVector<QPointF> &data)
{
   curveMap[id] = data;
   refreshPixmap();
}
```

La fonction `setCurveData()` définit les données de courbe pour un ID de courbe donné. S'il existe déjà une courbe portant le même ID dans `curveMap`, elle est remplacée par les nouvelles données de courbe ; sinon, la nouvelle courbe est simplement insérée. La variable membre `curveMap` est de type `QMap<int,QVector<QPointF>>`.

```
void Plotter::clearCurve(int id)
{
   curveMap.remove(id);
   refreshPixmap();
}
```

La fonction `clearCurve()` supprime la courbe spécifiée dans `curveMap`.

```
QSize Plotter::minimumSizeHint() const
{
   return QSize(6 * Margin, 4 * Margin);
}
```

La fonction `minimumSizeHint()` est similaire à `sizeHint()` ; tout comme `sizeHint()` spécifie la taille idéale d'un widget, `minimumSizeHint()` spécifie la taille minimale idéale d'un widget. Une disposition ne redimensionne jamais un widget en dessous de sa taille requise minimale.

La valeur que nous retournons est 300 _ 200 (vu que `Margin` est égal à 50) pour laisser une marge des quatre côtés et un peu d'espace pour le tracé. En dessous de cette taille, le tracé serait trop petit pour être utile.

```
QSize Plotter::sizeHint() const
{
   return QSize(12 * Margin, 8 * Margin);
}
```

Dans `sizeHint` nous retournons une taille "idéale" proportionnelle à la constante `Margin` et avec le même format d'image de 3:2 que nous avons utilisé pour `minimumSizeHint()`.

Ceci termine l'analyse des slots et des fonctions publiques de `Plotter`. Etudions à présent les gestionnaires d'événements protégés.

```
void Plotter::paintEvent(QPaintEvent * /* event */)
{
   QStylePainter painter(this);
   painter.drawPixmap(0, 0, pixmap);

   if (rubberBandIsShown) {
      painter.setPen(palette().light().color());
      painter.drawRect(rubberBandRect.normalized()
                                     .adjusted(0, 0, -1, -1));
   }

   if (hasFocus()) {
      QStyleOptionFocusRect option;
      option.initFrom(this);
      option.backgroundColor = palette().dark().color();
```

```
        painter.drawPrimitive(QStyle::PE_FrameFocusRect, option);
    }
}
```

Normalement, c'est dans `paintEvent()` que nous effectuons toutes les opérations de dessin. Cependant, dans notre exemple, nous avons dessiné tout le tracé auparavant dans `refreshPixmap()`, nous avons donc la possibilité d'afficher tout le tracé simplement en copiant le pixmap dans le widget à la position (0, 0).

Si le rectangle de sélection est visible, nous le dessinons au-dessus du tracé. Nous utilisons le composant "light" du groupe de couleurs actuel du widget comme couleur du crayon pour garantir un bon contraste avec l'arrière-plan "dark".Notez que nous dessinons directement sur le widget, nous ne touchons donc pas au pixmap hors champ. Utiliser `QRect::normalized()` vous assure que le rectangle de sélection présente une largeur et une hauteur positives (en changeant les coordonnées si nécessaire), et `adjusted()` réduit la taille du rectangle d'un pixel pour tenir compte de son contour d'un pixel.

Si le `Plotter` est activé, un rectangle "de focus" est dessiné au moyen de la fonction `drawPrimitive()` correspondant au style de widget, avec `QStyle::PE_FrameFocusRect` comme premier argument et `QStyleOptionFocusRect` comme second argument. Les options graphiques du rectangle de focus sont héritées du widget `Plotter` (par l'appel de `initFrom()`). La couleur d'arrière-plan doit être spécifiée explicitement.

Si vous voulez dessiner en utilisant le style actuel, vous pouvez appeler directement une fonction `QStyle`, par exemple,

```
style()->drawPrimitive(QStyle::PE_FrameFocusRect, &option, &painter,
                       this);
```

ou utiliser un `QStylePainter` au lieu d'un `QPainter` normal, comme nous avons procédé dans `Plotter`. Vous dessinez ainsi plus confortablement.

La fonction `QWidget::style()` retourne le style qui doit être utilisé pour dessiner le widget. Dans Qt, le style de widget est une sous-classe de `QStyle`. Les styles intégrés englobent `QWindowsStyle`, `QWindowsXPStyle`, `QMotifStyle`, `QCDEStyle`, `QMacStyle` et `QPlastiqueStyle`. Chacun de ces styles réimplémente les fonctions virtuelles dans `QStyle` afin d'adapter le dessin à la plate-forme pour laquelle le style est émulé. La fonction `drawPrimitive()` de `QStylePainter` appelle la fonction `QStyle` du même nom, qui peut être employée pour dessiner des "éléments primitifs" comme les panneaux, les boutons et les rectangles de focus. Le style de widget est généralement le même pour tous les widgets d'une application (`QApplication::style()`), mais vous pouvez l'adapter au cas par cas à l'aide de `QWidget::setStyle()`.

En dérivant `QStyle`, il est possible de définir un style personnalisé. Vous pouvez ainsi attribuer un aspect très particulier à une application ou une suite d'applications. Alors qu'il est habituellement recommandé d'adopter l'aspect et l'apparence natifs de la plate-forme cible, Qt offre une grande flexibilité si vous souhaitez intervenir dans ce domaine.

Les widgets intégrés de Qt se basent presque exclusivement sur `QStyle` pour se dessiner. C'est pourquoi ils ressemblent aux widgets natifs sur toutes les plates-formes prises en charge par Qt.

Les widgets personnalisés peuvent adopter le style courant soit en utilisant `QStyle` pour se tracer eux-mêmes, soit en employant les widgets Qt intégrés comme widgets enfants. S'agissant de `Plotter`, nous utilisons une combinaison des deux approches : le rectangle de focus est dessiné avec `QStyle` (*via* un `QStylePainter`) et les boutons Zoom In et Zoom Out sont des widgets Qt intégrés.

```
void Plotter::resizeEvent(QResizeEvent * /* event */)
{
   int x = width() - (zoomInButton->width()
                      + zoomOutButton->width() + 10);
   zoomInButton->move(x, 5);
   zoomOutButton->move(x + zoomInButton->width() + 5, 5);
   refreshPixmap();
}
```

Quand le widget `Plotter` est redimensionné, Qt déclenche un événement "`resize`". Ici, nous implémentons `resizeEvent()` pour placer les boutons Zoom In et Zoom Out en haut à droite du widget `Plotter`.

Nous déplaçons les boutons Zoom In et Zoom Out pour qu'ils soient côte à côte, séparés par un espace de 5 pixels et décalés de 5 pixels par rapport aux bords supérieur et droit du widget parent.

Si nous avions voulu que les boutons restent ancrés dans le coin supérieur gauche, dont les coordonnées sont (0, 0), nous les aurions simplement placés à cet endroit dans le constructeur de `Plotter`. Néanmoins, nous souhaitons assurer le suivi du coin supérieur droit, dont les coordonnées dépendent de la taille du widget. C'est pour cette raison qu'il est nécessaire de réimplémenter `resizeEvent()` et d'y définir la position des boutons.

Nous n'avons pas configuré les positions des boutons dans le constructeur de `Plotter`. Ce n'est pas un problème parce que Qt déclenche toujours un événement `resize` avant d'afficher un widget pour la première fois.

Plutôt que de réimplémenter `resizeEvent()` et de disposer les widgets enfants manuellement, nous aurions pu faire appel à un gestionnaire de disposition (par exemple, `QGridLayout`). L'utilisation d'une disposition aurait été un peu plus compliquée et aurait consommé davantage de ressources, mais les dispositions de droite à gauche aurait été mieux gérées, notamment pour des langues comme l'arabe et l'hébreu.

Nous terminons en invoquant `refreshPixmap()` pour redessiner le pixmap à sa nouvelle taille.

```
void Plotter::mousePressEvent(QMouseEvent *event)
{
   QRect rect(Margin, Margin,
              width() - 2 * Margin, height() - 2 * Margin);

   if (event->button() == Qt::LeftButton) {
      if (rect.contains(event->pos())) {
         rubberBandIsShown = true;
         rubberBandRect.setTopLeft(event->pos());
         rubberBandRect.setBottomRight(event->pos());
         updateRubberBandRegion();
```

```
            setCursor(Qt::CrossCursor);
        }
    }
}
```

Quand l'utilisateur appuie sur le bouton gauche de la souris, nous commençons à afficher un rectangle de sélection. Ceci implique de définir `rubberBandIsShown` en `true`, d'initialiser la variable membre `rubberBandRect` à la position actuelle du pointeur de la souris, de planifier un événement `paint` pour tracer le rectangle et de changer le pointeur de la souris pour afficher un pointeur à réticule.

La variable `rubberBandRect` est de type `QRect`. Un `QRect` peut être défini soit sous forme d'un quadruple `(x, y, width, height)` – où `(x, y)` est la position du coin supérieur gauche et `width _ height` correspond à la taille du rectangle – soit comme une paire de coordonnées supérieur-gauche et inférieur-droit. Dans ce cas, nous avons employé la représentation avec des paires de coordonnées. Nous définissons le point où l'utilisateur a cliqué à la fois comme étant le coin supérieur gauche et le coin inférieur droit. Puis nous appelons `updateRubberBandRegion()` pour forcer le rafraîchissement de l'affichage de la (toute petite) zone couverte par le rectangle de sélection.

Qt propose deux mécanismes pour contrôler la forme du pointeur de la souris :

- `QWidget::setCursor()` définit la forme du pointeur à utiliser quand la souris se place sur un widget particulier. Si aucun pointeur n'est configuré pour le widget, c'est le pointeur du widget parent qui est employé. Les widgets de haut niveau proposent par défaut un pointeur en forme de flèche.
- `QApplication::setOverrideCursor()` définit la forme du pointeur pour toute l'application, ignorant les pointeurs configurés par chaque widget jusqu'à ce que `restoreOverrideCursor()` soit invoquée.

Dans le Chapitre 4, nous avons appelé `QApplication::setOverrideCursor()` avec `Qt::WaitCursor` pour changer le pointeur de l'application en sablier.

```
void Plotter::mouseMoveEvent(QMouseEvent *event)
{
    if (rubberBandIsShown) {
        updateRubberBandRegion();
        rubberBandRect.setBottomRight(event->pos());
        updateRubberBandRegion();
    }
}
```

Quand l'utilisateur déplace le pointeur de la souris alors qu'il maintient le bouton gauche enfoncé, nous appelons d'abord `updateRubberBandRegion()` pour planifier un événement `paint` afin de redessiner la zone où se trouvait le rectangle de sélection, puis nous recalculons `rubberBandRect` pour tenir compte du déplacement de la souris, et enfin nous invoquons `updateRubberBandRegion()` une deuxième fois pour retracer la zone vers laquelle s'est

déplacé le rectangle de sélection. Ce rectangle est donc effectivement supprimé et redessiné aux nouvelles coordonnées.

Si l'utilisateur déplace la souris vers le haut ou la gauche, il est probable que le coin inférieur droit de `rubberBandRect` se retrouve au-dessus ou à gauche de son coin supérieur gauche. Si c'est le cas, `QRect` aura une largeur ou une hauteur négative. Nous avons utilisé `QRect::normalized()` dans `paintEvent()` pour nous assurer que les coordonnées supérieur-gauche et inférieur-droit sont ajustées de manière à ne pas avoir de largeur et de hauteur négatives.

```
void Plotter::mouseReleaseEvent(QMouseEvent *event)
{
   if ((event->button() == Qt::LeftButton) && rubberBandIsShown) {
      rubberBandIsShown = false;
      updateRubberBandRegion();
      unsetCursor();

      QRect rect = rubberBandRect.normalized();
      if (rect.width() < 4 || rect.height() < 4)
         return;
      rect.translate(-Margin, -Margin);

      PlotSettings prevSettings = zoomStack[curZoom];
      PlotSettings settings;
      double dx = prevSettings.spanX() / (width() - 2 * Margin);
      double dy = prevSettings.spanY() / (height() - 2 * Margin);

      settings.minX = prevSettings.minX + dx * rect.left();
      settings.maxX = prevSettings.minX + dx * rect.right();
      settings.minY = prevSettings.maxY - dy * rect.bottom();
      settings.maxY = prevSettings.maxY - dy * rect.top();
      settings.adjust();

      zoomStack.resize(curZoom + 1);
      zoomStack.append(settings);
      zoomIn();
   }
}
```

Quand l'utilisateur relâche le bouton gauche de la souris, nous supprimons le rectangle de sélection et nous restaurons le pointeur standard sous forme de flèche. Si le rectangle est au moins de 4×4, nous effectuons un zoom. Si le rectangle de sélection est plus petit, il est probable que l'utilisateur a cliqué sur le widget par erreur ou uniquement pour l'activer, nous ne faisons donc rien.

Le code permettant de zoomer est quelque peu complexe. C'est parce que nous traitons des coordonnées du widget et de celles du traceur en même temps. La plupart des tâches effectuées ici servent à convertir le `rubberBandRect`, pour transformer les coordonnées du widget en coordonnées du traceur. Une fois la conversion effectuée, nous invoquons

`PlotSettings::adjust()` pour arrondir les chiffres et trouver un nombre raisonnable de graduations pour chaque axe. Les Figures 5.10 et 5.11 illustrent la situation.

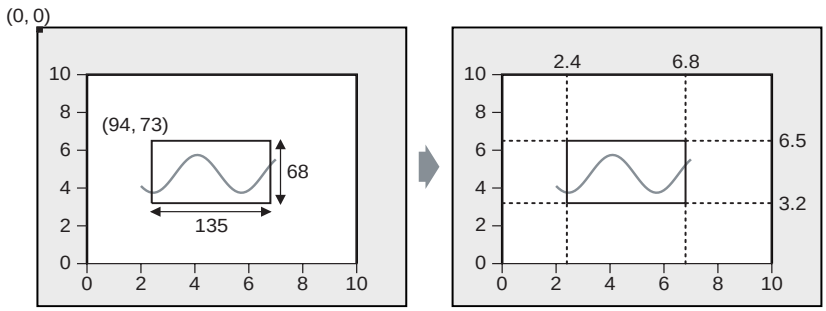


Figure 5.10
Convertir les coordonnées d'un rectangle de sélection du widget en coordonnées du traceur

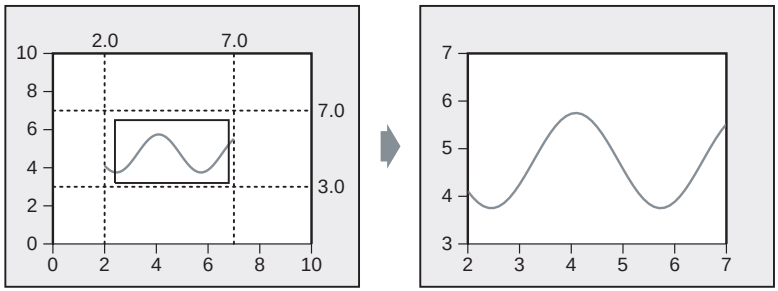


Figure 5.11
Ajuster les coordonnées du traceur et zoomer sur le rectangle de sélection

Puis nous zoomons. Pour zoomer, nous devons appuyer sur le nouveau `PlotSettings` que nous venons de calculer en haut de la pile de zoom et nous appelons `zoomIn()` qui se chargera de la tâche.

```
void Plotter::keyPressEvent(QKeyEvent *event)
{
    switch (event->key()) {
    case Qt::Key_Plus:
        zoomIn();
        break;
    case Qt::Key_Minus:
        zoomOut();
        break;
    case Qt::Key_Left:
        zoomStack[curZoom].scroll(-1, 0);
        refreshPixmap();
        break;
    case Qt::Key_Right:
        zoomStack[curZoom].scroll(+1, 0);
        refreshPixmap();
        break;
    case Qt::Key_Down:
```

```
            zoomStack[curZoom].scroll(0, -1);
            refreshPixmap();
            break;
        case Qt::Key_Up:
            zoomStack[curZoom].scroll(0, +1);
            refreshPixmap();
            break;
        default:
            QWidget::keyPressEvent(event);
        }
    }
```

Quand l'utilisateur appuie sur une touche et que le widget `Plotter` est actif, la fonction `keyPressEvent()` est invoquée. Nous la réimplémentons ici pour répondre à six touches : +, –, Haut, Bas, Gauche et Droite. Si l'utilisateur a appuyé sur une touche que nous ne gérons pas, nous appelons l'implémentation de la classe de base. Pour une question de simplicité, nous ignorons les touches de modification Maj, Ctrl et Alt, disponibles *via* `QKeyEvent::modifiers()`.

```
void Plotter::wheelEvent(QWheelEvent *event)
{
    int numDegrees = event->delta() / 8;
    int numTicks = numDegrees / 15;

    if (event->orientation() == Qt::Horizontal) {
        zoomStack[curZoom].scroll(numTicks, 0);
    } else {
        zoomStack[curZoom].scroll(0, numTicks);
    }
    refreshPixmap();
}
```

Les événements `wheel` se déclenchent quand la molette de la souris est actionnée. La majorité des souris ne proposent qu'une molette verticale, mais certaines sont équipées d'une molette horizontale. Qt prend en charge les deux types de molette. Les événements `wheel` sont transmis au widget actif. La fonction `delta()` retourne la distance parcourue par la molette en huitièmes de degré. Les souris proposent habituellement une plage de 15 degrés. Dans notre exemple, nous faisons défiler le nombre de graduations demandées en modifiant l'élément le plus haut dans la pile de zoom et nous mettons à jour l'affichage au moyen de `refreshPixmap()`.

Nous utilisons la molette de la souris le plus souvent pour faire dérouler une barre de défilement. Quand nous employons `QScrollArea` (traité dans le Chapitre 6) pour proposer des barres de défilement, `QScrollArea` gère automatiquement les événements liés à la molette de la souris, nous n'avons donc pas à réimplémenter `wheelEvent()` nous-mêmes.

Ceci achève l'implémentation des gestionnaires d'événements. Passons maintenant en revue les fonctions privées.

```
void Plotter::updateRubberBandRegion()
{
```

```
    QRect rect = rubberBandRect.normalized();
    update(rect.left(), rect.top(), rect.width(), 1);
    update(rect.left(), rect.top(), 1, rect.height());
    update(rect.left(), rect.bottom(), rect.width(), 1);
    update(rect.right(), rect.top(), 1, rect.height());
}
```

La fonction `updateRubberBand()` est appelée depuis `mousePressEvent()`, `mouseMoveEvent()` et `mouseReleaseEvent()` pour effacer ou redessiner le rectangle de sélection. Elle est constituée de quatre appels de `update()` qui planifient un événement `paint` pour les quatre petites zones rectangulaires couvertes par le rectangle de sélection (deux lignes verticales et deux lignes horizontales). Qt propose la classe `QRubberBand` pour dessiner des rectangles de sélection, mais dans ce cas, l'écriture du code permet de mieux contrôler l'opération.

```
void Plotter::refreshPixmap()
{
    pixmap = QPixmap(size());
    pixmap.fill(this, 0, 0);

    QPainter painter(&pixmap);
    painter.initFrom(this);
    drawGrid(&painter);
    drawCurves(&painter);
    update();
}
```

La fonction `refreshPixmap()` redessine le tracé sur le pixmap hors champ et met l'affichage à jour. Nous redimensionnons le pixmap de sorte qu'il ait la même taille que le widget et nous le remplissons avec la couleur d'effacement du widget. Cette couleur correspond au composant "dark" de la palette en raison de l'appel de `setBackgroundRole()` dans le constructeur de `Plotter`. Si l'arrière-plan n'est pas uni, `QPixmap::fill()` doit connaître la position du pixmap dans le widget pour aligner correctement le motif de couleur. Dans notre cas, le pixmap correspond à la totalité du widget, nous spécifions donc la position (0, 0).

Nous créons ensuite un `QPainter` pour dessiner sur le pixmap. L'appel de `initFrom()` définit le crayon, l'arrière-plan et la police pour qu'ils soient identiques à ceux du widget `Plotter`. Puis nous invoquons `drawGrid()` et `drawCurves()` pour réaliser le dessin. Nous appelons enfin `update()` pour planifier un événement `paint` pour la totalité du widget. Le pixmap est copié dans le widget dans la fonction `paintEvent()`.

```
void Plotter::drawGrid(QPainter *painter)
{
    QRect rect(Margin, Margin,
               width() - 2 * Margin, height() - 2 * Margin);
    if (!rect.isValid())
        return;

    PlotSettings settings = zoomStack[curZoom];
    QPen quiteDark = palette().dark().color().light();
    QPen light = palette().light().color();
```

```
    for (int i = 0; i <= settings.numXTicks; ++i) {
        int x = rect.left() + (i * (rect.width() - 1)
                               / settings.numXTicks);
        double label = settings.minX + (i * settings.spanX()
                                        / settings.numXTicks);
        painter->setPen(quiteDark);
        painter->drawLine(x, rect.top(), x, rect.bottom());
        painter->setPen(light);
        painter->drawLine(x, rect.bottom(), x, rect.bottom() + 5);
        painter->drawText(x - 50, rect.bottom() + 5, 100, 15,
                          Qt::AlignHCenter | Qt::AlignTop,
                          QString::number(label));
    }
    for (int j = 0; j <= settings.numYTicks; ++j) {
        int y = rect.bottom() - (j * (rect.height() - 1)
                                 / settings.numYTicks);
        double label = settings.minY + (j * settings.spanY()
                                        / settings.numYTicks);
        painter->setPen(quiteDark);
        painter->drawLine(rect.left(), y, rect.right(), y);
        painter->setPen(light);
        painter->drawLine(rect.left() - 5, y, rect.left(), y);
        painter->drawText(rect.left() - Margin, y - 10, Margin - 5, 20,
                          Qt::AlignRight | Qt::AlignVCenter,
                          QString::number(label));
    }
    painter->drawRect(rect.adjusted(0, 0, -1, -1));
}
```

La fonction `drawGrid()` dessine la grille derrière les courbes et les axes. La zone dans laquelle nous dessinons la grille est spécifiée par `rect`. Si le widget n'est pas assez grand pour s'adapter au graphique, nous retournons immédiatement.

La première boucle `for` trace les lignes verticales de la grille et les graduations sur l'axe *x*. La seconde boucle `for` trace les lignes horizontales de la grille et les graduations sur l'axe *y*. A la fin, nous dessinons un rectangle le long des marges. La fonction `drawText()` dessine les numéros correspondants aux graduations sur les deux axes.

Les appels de `drawText()` ont la syntaxe suivante :

```
painter->drawText(x, y, width, height, alignment, text);
```

où `(x, y, width, height)` définit un rectangle, `alignment` la position du texte dans ce rectangle et `text` le texte à dessiner.

```
void Plotter::drawCurves(QPainter *painter)
{
    static const QColor colorForIds[6] = {
        Qt::red, Qt::green, Qt::blue, Qt::cyan, Qt::magenta, Qt::yellow
    };
    PlotSettings settings = zoomStack[curZoom];
    QRect rect(Margin, Margin,
               width() - 2 * Margin, height() - 2 * Margin);
```

```
    if (!rect.isValid())
        return;

    painter->setClipRect(rect.adjusted(+1, +1, -1, -1));

    QMapIterator<int, QVector<QPointF> > i(curveMap);
    while (i.hasNext()) {
        i.next();

        int id = i.key();
        const QVector<QPointF> &data = i.value();
        QPolygonF polyline(data.count());

        for (int j = 0; j < data.count(); ++j) {
            double dx = data[j].x() - settings.minX;
            double dy = data[j].y() - settings.minY;
            double x = rect.left() + (dx * (rect.width() - 1)
                                       / settings.spanX());
            double y = rect.bottom() - (dy * (rect.height() - 1)
                                         / settings.spanY());
            polyline[j] = QPointF(x, y);
        }
        painter->setPen(colorForIds[uint(id) % 6]);
        painter->drawPolyline(polyline);
    }
}
```

La fonction `drawCurves()` dessine les courbes au-dessus de la grille. Nous commençons par appeler `setClipRect()` pour définir la zone d'action de `QPainter` comme égale au rectangle qui contient les courbes (excepté les marges et le cadre autour du graphique). `QPainter` ignorera ensuite les opérations de dessin sur les pixels situés en dehors de cette zone.

Puis, nous parcourons toutes les courbes à l'aide d'un itérateur de style Java, et pour chacune d'elles, nous parcourons les `QPointF` dont elle est constituée. La fonction `key()` donne l'ID de la courbe et la fonction `value()` donne les données de courbe correspondantes comme un `QVector<QPointF>`. La boucle interne `for` convertit chaque `QPointF` pour transformer les coordonnées du traceur en coordonnées du widget et les stocke dans la variable `polyline`.

Une fois que nous avons converti tous les points d'une courbe en coordonnées du widget, nous déterminons la couleur de crayon pour la courbe (en utilisant un des ensembles de couleurs prédéfinies) et nous appelons `drawPolyline()` pour tracer une ligne qui passe par tous les points de cette dernière.

Voici la classe `Plotter` terminée. Tout ce qui reste, ce sont quelques fonctions dans `PlotSettings`.

```
PlotSettings::PlotSettings()
{
    minX = 0.0;
    maxX = 10.0;
    numXTicks = 5;
```

```
    minY = 0.0;
    maxY = 10.0;
    numYTicks = 5;
}
```

Le constructeur de `PlotSettings` initialise les deux axes avec une plage de 0 à 10 en 5 graduations.

```
void PlotSettings::scroll(int dx, int dy)
{
    double stepX = spanX() / numXTicks;
    minX += dx * stepX;
    maxX += dx * stepX;

    double stepY = spanY() / numYTicks;
    minY += dy * stepY;
    maxY += dy * stepY;
}
```

La fonction `scroll()` incrémente (ou décrémente) `minX`, `maxX`, `minY` et `maxY` de la valeur de l'intervalle entre deux graduations multipliée par un nombre donné. Cette fonction est utilisée pour implémenter le défilement dans `Plotter::keyPressEvent()`.

```
void PlotSettings::adjust()
{
    adjustAxis(minX, maxX, numXTicks);
    adjustAxis(minY, maxY, numYTicks);
}
```

La fonction `adjust()` est invoquée dans `mouseReleaseEvent()` pour arrondir les valeurs `minX`, `maxX`, `minY` et `maxY` en valeurs "conviviales" et pour déterminer le bon nombre de graduations pour chaque axe. La fonction privée `adjustAxis()` s'exécute sur un axe à la fois.

```
void PlotSettings::adjustAxis(double &min, double &max,
                              int &numTicks)
{
    const int MinTicks = 4;
    double grossStep = (max - min) / MinTicks;
    double step = pow(10.0, floor(log10(grossStep)));

    if (5 * step < grossStep) {
       step *= 5;
    } else if (2 * step < grossStep) {
    step *= 2;
    }

    numTicks = int(ceil(max / step) - floor(min / step));
    if (numTicks < MinTicks)
        numTicks = MinTicks;
    min = floor(min / step) * step;
    max = ceil(max / step) * step;
}
```

La fonction `adjustAxis()` convertit ses paramètres `min` et `max` en nombres "conviviaux" et définit son paramètre `numTicks` en nombre de graduations qu'elle calcule comme étant appropriées pour la plage `[min, max]` donnée. Vu que `adjustAxis()` a besoin de modifier les variables réelles (`minX`, `maxX`, `numXTicks`, etc.) et pas uniquement des copies, ses paramètres sont des références non-const.

Le code de `adjustAxis()` est principalement consacré à déterminer une valeur adéquate pour l'intervalle entre deux graduations ("l'échelon"). Pour obtenir des nombres convenables sur l'axe, nous devons sélectionner l'échelon avec soin. Par exemple, une valeur de 3,8 engendrerait des multiples de 3,8 sur un axe, ce qui n'est pas très significatif pour les utilisateurs. Pour les axes avec une notation décimale, des valeurs d'échelons "conviviales" sont des chiffres de la forme 10^n, 2×10^n ou 5×10^n.

Nous commençons par calculer "l'échelon brut," une sorte de valeur maximum pour l'échelon. Puis nous recherchons le nombre correspondant sous la forme 10^n qui est inférieur ou égal à l'échelon brut. Pour ce faire, nous prenons le logarithme décimal de l'échelon brut, en arrondissant cette valeur vers le bas pour obtenir un nombre entier, puis en ajoutant 10 à la puissance de ce chiffre arrondi. Par exemple, si l'échelon brut est de 236, nous calculons log 236 = 2,37291… ; puis nous l'arrondissons vers le bas pour aboutir à 2 et nous obtenons $10^2 = 100$ comme valeur d'échelon sous la forme 10^n.

Une fois que la première valeur d'échelon est déterminée, nous pouvons l'utiliser pour calculer les deux autres candidats : 2×10^n et 5×10^n. Dans l'exemple ci-dessus, les deux autres candidats sont 200 et 500. Le candidat 500 est supérieur à l'échelon brut, nous n'avons donc pas la possibilité de l'employer. Mais 200 est inférieur à 236, nous utilisons ainsi 200 comme taille d'échelon dans cet exemple.

Il est assez facile de calculer `numTicks`, `min` et `max` à partir de la valeur d'échelon. La nouvelle valeur `min` est obtenue en arrondissant la valeur `min` d'origine vers le bas vers le multiple le plus proche de l'échelon, et la nouvelle valeur `max` est obtenue en arrondissant vers le haut vers le multiple le plus proche de l'échelon. La nouvelle valeur `numTicks` correspond au nombre d'intervalles entre les valeurs `min` et `max` arrondies. Par exemple, si `min` est égal à 240 et `max` à 1184 au moment de la saisie de la fonction, la nouvelle plage devient [200, 1200], avec cinq graduations.

Cet algorithme donnera des résultats optimaux dans certains cas. Un algorithme plus sophistiqué est décrit dans l'article "Nice Numbers for Graph Labels" de Paul S. Heckbert publié dans *Graphics Gems* (ISBN 0-12-286166-3).

Ce chapitre achève la Partie I. Il vous a expliqué comment personnaliser un widget Qt existant et comment générer un widget en partant de zéro à l'aide de `QWidget` comme classe de base. Nous avons aussi vu comment composer un widget à partir de widgets existants dans le Chapitre 2 et nous allons explorer ce thème plus en détail dans le Chapitre 6.

A ce stade, vous en savez suffisamment pour écrire des applications GUI complètes avec Qt. Dans les Parties II et III, nous étudierons Qt en profondeur pour pouvoir profiter de toute la puissance de ce framework.

II

Qt : niveau intermédiaire

6 *Gestion des dispositions*
7 *Traitement des événements*
8 *Graphiques 2D et 3D*
9 *Glisser-déposer*
10 *Classes d'affichage d'éléments*
11 *Classes conteneur*
12 *Entrées/Sorties*
13 *Les bases de données*
14 *Gestion de réseau*
15 *XML*
16 *Aide en ligne*

6

Gestion des dispositions

Au sommaire de ce chapitre

- ✔ Disposer des widgets sur un formulaire
- ✔ Dispositions empilées
- ✔ Séparateurs
- ✔ Zones déroulantes
- ✔ Widgets et barres d'outils ancrables
- ✔ MDI (Multiple Document Interface)

Chaque widget placé dans un formulaire doit se voir attribuer une taille et une position appropriées. Qt propose plusieurs classes qui disposent les widgets dans un formulaire : `QHBoxLayout`, `QVBoxLayout`, `QGridLayout` et `QStackLayout`. Ces classes sont si pratiques et faciles à utiliser que presque tous les développeurs Qt s'en servent, soit directement dans du code source, soit par le biais du *Qt Designer*.

Il existe une autre raison d'employer les classes de disposition (layout) de Qt : elles garantissent que les formulaires s'adaptent automatiquement aux diverses polices, langues et plates-formes. Si l'utilisateur modifie les paramètres de police du système, les formulaires de l'application répondront immédiatement en se redimensionnant eux-mêmes si nécessaire. Si vous traduisez l'interface utilisateur de l'application en d'autres langues, les classes de disposition prennent en compte le contenu traduit des widgets pour éviter toute coupure de texte.

`QSplitter`, `QScrollArea`, `QMainWindow` et `QWorkspace` sont d'autres classes qui se chargent de gérer la disposition. Le point commun de ces classes c'est qu'elles procurent une disposition très flexible sur laquelle l'utilisateur peut agir. Par exemple, `QSplitter` propose un séparateur que l'utilisateur peut faire glisser pour redimensionner les widgets, et `QWorkspace` prend en charge MDI (multiple document interface), un moyen d'afficher plusieurs documents simultanément dans la fenêtre principale d'une application. Etant donné qu'elles sont souvent utilisées comme des alternatives aux classes de disposition, elles sont aussi présentées dans ce chapitre.

Disposer des widgets sur un formulaire

Il y a trois moyens de gérer la disposition des widgets enfants dans un formulaire : le positionnement absolu, la disposition manuelle et les gestionnaires de disposition. Nous allons étudier chacun d'eux à tour de rôle, en nous basant sur la boîte de dialogue Find File illustrée en Figure 6.1.

Figure 6.1
La boîte de dialogue Find File

Le positionnement absolu est le moyen le plus rudimentaire de disposer des widgets. Il suffit d'assigner dans du code des tailles et des positions aux widgets enfants du formulaire et une taille fixe au formulaire. Voici à quoi ressemble le constructeur de `FindFileDialog` avec le positionnement absolu :

```
FindFileDialog::FindFileDialog(QWidget *parent)
   : QDialog(parent)
{
   ...
   namedLabel->setGeometry(9, 9, 50, 25);
   namedLineEdit->setGeometry(65, 9, 200, 25);
   lookInLabel->setGeometry(9, 40, 50, 25);
   lookInLineEdit->setGeometry(65, 40, 200, 25);
```

```
    subfoldersCheckBox->setGeometry(9, 71, 256, 23);
    tableWidget->setGeometry(9, 100, 256, 100);
    messageLabel->setGeometry(9, 206, 256, 25);
    findButton->setGeometry(271, 9, 85, 32);
    stopButton->setGeometry(271, 47, 85, 32);
    closeButton->setGeometry(271, 84, 85, 32);
    helpButton->setGeometry(271, 199, 85, 32);

    setWindowTitle(tr("Find Files or Folders"));
    setFixedSize(365, 240);
}
```

Le positionnement absolu présente de nombreux inconvénients :

- L'utilisateur ne peut pas redimensionner la fenêtre.
- Une partie du texte peut être coupée si l'utilisateur choisit une police trop grande ou si l'application est traduite dans une autre langue.
- Les widgets peuvent présenter des tailles inadaptées pour certains styles.
- Les positions et les tailles doivent être calculées manuellement. Cette méthode est fastidieuse et sujette aux erreurs ; de plus, elle complique la maintenance.

L'alternative au positionnement absolu est la disposition manuelle. Avec cette technique, les widgets ont toujours des positions absolues données, mais leurs tailles sont proportionnelles à la taille de la fenêtre au lieu d'être totalement codées. Il convient donc de réimplémenter la fonction `resizeEvent()` du formulaire pour définir les géométries de ses widgets enfants :

```
FindFileDialog::FindFileDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    setMinimumSize(265, 190);
    resize(365, 240);
}

void FindFileDialog::resizeEvent(QResizeEvent * /* event */)
{
    int extraWidth = width() - minimumWidth();
    int extraHeight = height() - minimumHeight();

    namedLabel->setGeometry(9, 9, 50, 25);
    namedLineEdit->setGeometry(65, 9, 100 + extraWidth, 25);
    lookInLabel->setGeometry(9, 40, 50, 25);
    lookInLineEdit->setGeometry(65, 40, 100 + extraWidth, 25);
    subfoldersCheckBox->setGeometry(9, 71, 156 + extraWidth, 23);

    tableWidget->setGeometry(9, 100, 156 + extraWidth,
                             50 + extraHeight);
    messageLabel->setGeometry(9, 156 + extraHeight, 156 + extraWidth,
                              25);
    findButton->setGeometry(171 + extraWidth, 9, 85, 32);
    stopButton->setGeometry(171 + extraWidth, 47, 85, 32);
```

```
    closeButton->setGeometry(171 + extraWidth, 84, 85, 32);
    helpButton->setGeometry(171 + extraWidth, 149 + extraHeight, 85,
                            32);
}
```

Dans le constructeur de `FindFileDialog`, nous configurons la taille minimale du formulaire en 265×190 et la taille initiale en 365×240. Dans le gestionnaire `resizeEvent()`, nous accordons de l'espace supplémentaire aux widgets qui veulent s'agrandir. Nous sommes ainsi certains que le formulaire se met à l'échelle quand l'utilisateur le redimensionne, comme illustré en Figure 6.2.

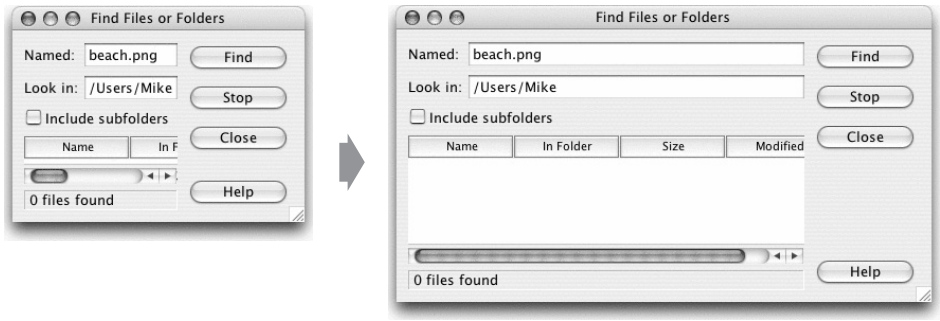


Figure 6.2
Redimensionner une boîte de dialogue redimensionnable

Tout comme le positionnement absolu, la disposition manuelle oblige le programmeur à calculer beaucoup de constantes codées. Ecrire du code de cette manière se révèle pénible, notamment si la conception change. De plus, le texte court toujours le risque d'être coupé. Nous pouvons éviter ce problème en tenant compte des tailles requises des widgets enfants, mais cela compliquerait encore plus le code.

La solution la plus pratique pour disposer des widgets sur un formulaire consiste à utiliser les gestionnaires de disposition de Qt. Ces gestionnaires proposent des valeurs par défaut raisonnables pour chaque type de widget et tiennent compte de la taille requise de chacun d'eux, qui dépend de la police, du style et du contenu du widget. Ces gestionnaires respectent également des dimensions minimales et maximales, et ajustent automatiquement la disposition en réponse à des changements de police ou de contenu et à un redimensionnement de la fenêtre.

Les trois gestionnaires de disposition les plus importants sont `QHBoxLayout`, `QVBoxLayout` et `QGridLayout`. Ces classes héritent de `QLayout`, qui fournit le cadre de base des dispositions. Ces trois classes sont totalement prises en charge par le *Qt Designer* et peuvent aussi être utilisées directement dans du code.

Voici le code de `FindFileDialog` avec des gestionnaires de disposition :

```
FindFileDialog::FindFileDialog(QWidget *parent)
```

```
    : QDialog(parent)
{
    ...
    QGridLayout *leftLayout = new QGridLayout;
    leftLayout->addWidget(namedLabel, 0, 0);
    leftLayout->addWidget(namedLineEdit, 0, 1);
    leftLayout->addWidget(lookInLabel, 1, 0);
    leftLayout->addWidget(lookInLineEdit, 1, 1);
    leftLayout->addWidget(subfoldersCheckBox, 2, 0, 1, 2);
    leftLayout->addWidget(tableWidget, 3, 0, 1, 2);
    leftLayout->addWidget(messageLabel, 4, 0, 1, 2);

    QVBoxLayout *rightLayout = new QVBoxLayout;
    rightLayout->addWidget(findButton);
    rightLayout->addWidget(stopButton);
    rightLayout->addWidget(closeButton);
    rightLayout->addStretch();
    rightLayout->addWidget(helpButton);

    QHBoxLayout *mainLayout = new QHBoxLayout;
    mainLayout->addLayout(leftLayout);
    mainLayout->addLayout(rightLayout);
    setLayout(mainLayout);

    setWindowTitle(tr("Find Files or Folders"));
}
```

La disposition est gérée par `QHBoxLayout`, `QGridLayout` et `QVBoxLayout`. `QGridLayout` à gauche et `QVBoxLayout` à droite sont placés côte à côte par le `QHBoxLayout` externe. Les marges autour de la boîte de dialogue et l'espace entre les widgets enfants présentent des valeurs par défaut en fonction du style de widget ; elles peuvent être modifiées grâce à `QLayout::setMargin()` et `QLayout::setSpacing()`.

La même boîte de dialogue aurait pu être créée visuellement dans le *Qt Designer* en plaçant les widgets enfants à leurs positions approximatives, en sélectionnant ceux qui doivent être disposés ensemble et en cliquant sur Form > Lay Out Horizontally, Form > Lay Out Vertically ou Form > Lay Out in a Grid. Nous avons employé cette approche dans le Chapitre 2 pour créer les boîtes de dialogue Go-to-Cell et Sort de l'application Spreadsheet.

Utiliser `QHBoxLayout` et `QVBoxLayout` est plutôt simple mais l'utilisation de `QGridLayout` se révèle un peu plus complexe. `QGridLayout` se base sur une grille de cellules à deux dimensions. Le `QLabel` dans le coin supérieur gauche de la disposition se trouve à la position (0, 0) et le `QLineEdit` correspondant se situe à la position (0, 1). Le `QCheckBox` s'étend sur deux colonnes ; il occupe les cellules aux positions (2, 0) et (2, 1). Les `QTreeWidget` et `QLabel` en dessous prennent aussi deux colonnes (voir Figure 6.3). Les appels de `addWidget()` ont la syntaxe suivante :

```
layout->addWidget(widget, row, column, rowSpan, columnSpan);
```

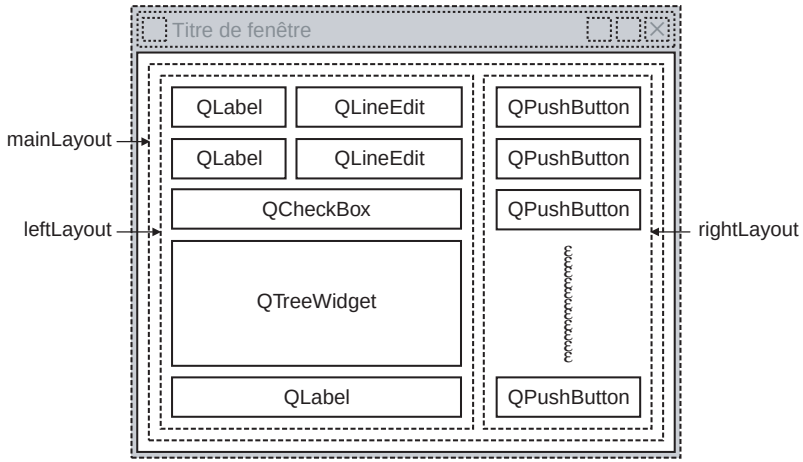


Figure 6.3
La disposition de la boîte de dialogue Find File

Dans ce cas, `widget` est le widget enfant à insérer dans la disposition, `(row, column)` est la cellule en haut à gauche occupée par le widget, `rowSpan` correspond au nombre de lignes occupées par le widget et `columnSpan` est le nombre de colonnes occupées par le widget. S'ils sont omis, les paramètres `rowSpan` et `columnSpan` ont la valeur 1 par défaut.

L'appel de `addStretch()` ordonne au gestionnaire de disposition d'utiliser l'espace à cet endroit dans la disposition. En ajoutant un élément d'étirement, nous avons demandé au gestionnaire de disposition de placer tout espace excédentaire entre les boutons Close et Help. Dans le *Qt Designer*, nous pouvons aboutir au même effet en insérant un élément d'espacement. Les éléments d'espacement apparaissent dans le *Qt Designer* sous forme de "ressorts" bleus.

Utiliser des gestionnaires de disposition présente des avantages supplémentaires par rapport à ceux décrits jusque là. Si nous ajoutons ou supprimons un widget dans une disposition, celle-ci s'adaptera automatiquement à la nouvelle situation. Il en va de même si nous invoquons `hide()` ou `show()` sur un widget enfant. Si la taille requise d'un widget enfant change, la disposition sera automatiquement corrigée, en tenant compte de cette nouvelle taille. En outre, les gestionnaires de disposition définissent automatiquement une taille minimale pour le formulaire, en fonction des tailles minimales et des tailles requises des widgets enfants de ce dernier.

Dans les exemples donnés jusqu'à présent, nous avons simplement placé les widgets dans des dispositions et utilisé des éléments d'espacement pour combler tout espace excédentaire. Dans certains cas, ce n'est pas suffisant pour que la disposition ressemble exactement à ce que nous voulons. Nous pouvons donc ajuster la disposition en changeant les stratégies de taille (règles auxquelles la taille est soumise, voir chapitre précédent) et les tailles requises des widgets à disposer.

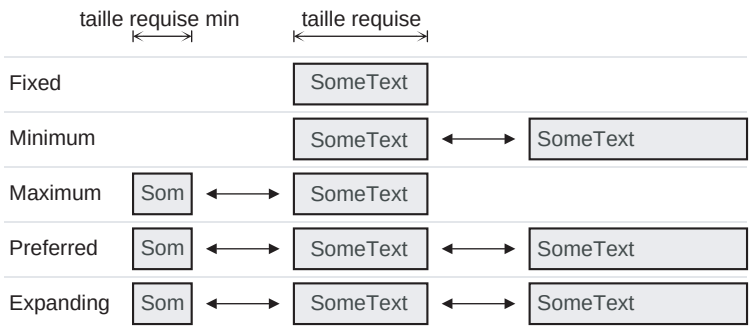
Grâce à la stratégie de taille d'un widget, le système de disposition sait comment ce widget doit être étiré ou rétréci. Qt propose des stratégies par défaut raisonnables pour tous ses widgets intégrés. Cependant, puisqu'il n'existe pas de valeur par défaut qui pourrait tenir compte de toutes les dispositions possibles, il est courant que les développeurs modifient les

stratégies pour un ou deux widgets dans un formulaire. Un `QSizePolicy` possède un composant horizontal et un vertical. Voici les valeurs les plus utiles :

- `Fixed` signifie que le widget ne peut pas être rétréci ou étiré. Le widget conserve toujours sa taille requise.
- `Minimum` signifie que la taille requise d'un widget correspond à sa taille minimale. Le widget ne peut pas être rétréci en dessous de la taille requise, mais il peut s'agrandir pour combler l'espace disponible si nécessaire.
- `Maximum` signifie que la taille requise d'un widget correspond à sa taille maximale. Le widget peut être rétréci jusqu'à sa taille requise minimum.
- `Preferred` signifie que la taille requise d'un widget correspond à sa taille favorite, mais que le widget peut toujours être rétréci ou étiré si nécessaire.
- `Expanding` signifie que le widget peut être rétréci ou étiré, mais qu'il préfère être agrandi.

La Figure 6.4 récapitule la signification des différentes stratégies, en utilisant un `QLabel` affichant le texte "Some Text" comme exemple.

Figure 6.4
La signification des différentes stratégies de taille



Dans la figure, `Preferred` et `Expanding` donnent le même résultat. Où se situe la différence ? Quand un formulaire qui contient les widgets `Preferred` et `Expanding` est redimensionné, l'espace supplémentaire est attribué aux widgets `Expanding`, alors que les widgets `Preferred` conservent leur taille requise.

Il existe deux autres stratégies : `MinimumExpanding` et `Ignored`. La première était nécessaire dans quelques rares cas dans les versions antérieures de Qt, mais elle ne présente plus d'intérêt ; la meilleure approche consiste à utiliser `Expanding` et à réimplémenter `minimumSizeHint()` de façon appropriée. La seconde est similaire à `Expanding`, sauf qu'elle ignore la taille requise et la taille requise minimum du widget.

En plus des composants verticaux et horizontaux de la stratégie, la classe `QSizePolicy` stocke un facteur d'étirement horizontal et vertical. Ces facteurs d'étirement peuvent être utilisés pour indiquer que les divers widgets enfants doivent s'étirer à différents niveaux quand le formulaire s'agrandit. Par exemple, si nous avons un `QTreeWidget` au-dessus d'un `QTextEdit` et que nous voulons que le `QTextEdit` soit deux fois plus grand que le `QTreeWidget`, nous avons la

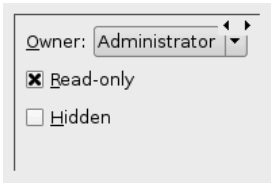
possibilité de définir un facteur d'étirement vertical de `QTextEdit` de 2 et un facteur d'étirement vertical de `QTreeWidget` de 1.

Cependant, un autre moyen d'influencer une disposition consiste à configurer une taille minimale ou maximale, ou une taille fixe pour les widgets enfants. Le gestionnaire de disposition respectera ces contraintes lorsqu'il disposera les widgets. Et si ce n'est pas suffisant, nous pouvons toujours dériver de la classe du widget enfant et réimplémenter `sizeHint()` pour obtenir la taille requise dont nous avons besoin.

Dispositions empilées

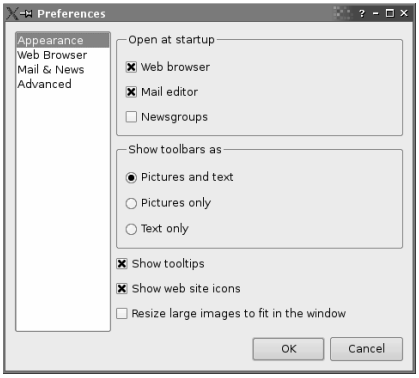
La classe `QStackedLayout` dispose un ensemble de widgets enfants, ou "pages," et n'en affiche qu'un seul à la fois, en masquant les autres à l'utilisateur. `QStackedLayout` est invisible en soi et l'utilisateur n'a aucun moyen de changer une page. Les petites flèches et le cadre gris foncé dans la Figure 6.5 sont fournis par le *Qt Designer* pour faciliter la conception avec la disposition. Pour des questions pratiques, Qt inclut également un `QStackedWidget` qui propose un `QWidget` avec un `QStackedLayout` intégré.

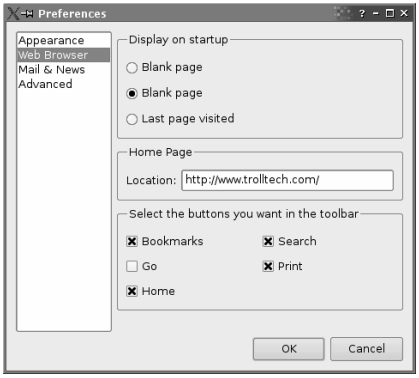
Figure 6.5
QStackedLayout



Les pages sont numérotées en commençant à 0. Pour afficher un widget enfant spécifique, nous pouvons appeler `setCurrentIndex()` avec un numéro de page. Le numéro de page d'un widget enfant est disponible grâce à `indexOf()`.

Figure 6.6
Deux pages de la boîte de dialogue Preferences





La boîte de dialogue Preferences illustrée en Figure 6.6 est un exemple qui utilise `QStackedLayout`. Elle est constituée d'un `QListWidget` à gauche et d'un `QStackedLayout` à droite. Chaque élément dans `QListWidget` correspond à une page différente dans `QStackedLayout`. Voici le code du constructeur de la boîte de dialogue :

```
PreferenceDialog::PreferenceDialog(QWidget *parent)
   : QDialog(parent)
{
   ...
   listWidget = new QListWidget;
   listWidget->addItem(tr("Appearance"));
   listWidget->addItem(tr("Web Browser"));
   listWidget->addItem(tr("Mail & News"));
   listWidget->addItem(tr("Advanced"));

   stackedLayout = new QStackedLayout;
   stackedLayout->addWidget(appearancePage);
   stackedLayout->addWidget(webBrowserPage);
   stackedLayout->addWidget(mailAndNewsPage);
   stackedLayout->addWidget(advancedPage);
   connect(listWidget, SIGNAL(currentRowChanged(int)),
           stackedLayout, SLOT(setCurrentIndex(int)));
   ...
   listWidget->setCurrentRow(0);
}
```

Nous créons un `QListWidget` et nous l'alimentons avec les noms des pages. Nous créons ensuite un `QStackedLayout` et nous invoquons `addWidget()` pour chaque page. Nous connectons le signal `currentRowChanged(int)` du widget liste à `setCurrentIndex(int)` de la disposition empilée pour implémenter le changement de page, puis nous appelons `setCurrentRow()` sur le widget liste à la fin du constructeur pour commencer à la page 0.

Ce genre de formulaire est aussi très facile à créer avec le *Qt Designer :*

1. créez un nouveau formulaire en vous basant sur les modèles "Dialog" ou "Widget" ;
2. ajoutez un `QListWidget` et un `QStackedWidget` au formulaire ;
3. remplissez chaque page avec des widgets enfants et des dispositions ;

   (Pour créer une nouvelle page, cliquez du bouton droit et sélectionnez Insert Page ; pour changer de page, cliquez sur la petite flèche gauche ou droite située en haut à droite du `QStackedWidget`)
4. disposez les widgets côte à côte grâce à une disposition horizontale ;
5. connectez le signal `currentRowChanged(int)` du widget liste au slot `setCurrentIndex(int)` du widget empilé ;
6. définissez la valeur de la propriété `currentRow` du widget liste en 0.

Etant donné que nous avons implémenté le changement de page en utilisant des signaux et des slots prédéfinis, la boîte de dialogue présentera le bon comportement quand elle sera prévisualisée dans le *Qt Designer.*

Séparateurs

`QSplitter` est un widget qui comporte d'autres widgets. Les widgets dans un séparateur sont séparés par des poignées. Les utilisateurs peuvent changer les tailles des widgets enfants du séparateur en faisant glisser ces poignées. Les séparateurs peuvent souvent être utilisés comme une alternative aux gestionnaires de disposition, pour accorder davantage de contrôle à l'utilisateur.

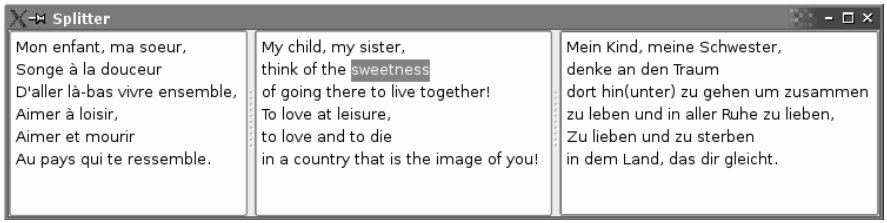


Figure 6.7
L'application Splitter

Les widgets enfants d'un `QSplitter` sont automatiquement placés côte à côte (ou un en-dessous de l'autre) dans l'ordre dans lequel ils sont créés, avec des barres de séparation entre les widgets adjacents. Voici le code permettant de créer la fenêtre illustrée en Figure 6.7 :

```
int main(int argc, char *argv[])
{
   QApplication app(argc, argv);

   QTextEdit *editor1 = new QTextEdit;
   QTextEdit *editor2 = new QTextEdit;
   QTextEdit *editor3 = new QTextEdit;

   QSplitter splitter(Qt::Horizontal);
   splitter.addWidget(editor1);
   splitter.addWidget(editor2);
   splitter.addWidget(editor3);
   ...
   splitter.show();
   return app.exec();
}
```

L'exemple est constitué de trois `QTextEdit` disposés horizontalement par un widget `QSplitter`. Contrairement aux gestionnaires de disposition qui se contentent d'organiser les widgets enfants d'un formulaire et ne proposent aucune représentation visuelle, `QSplitter` hérite de `QWidget` et peut être utilisé comme n'importe quel autre widget.

Vous obtenez des dispositions complexes en imbriquant des `QSplitter` horizontaux et verticaux. Par exemple, l'application Mail Client présentée en Figure 6.9 consiste en un `QSplitter` horizontal qui contient un `QSplitter` vertical sur sa droite.

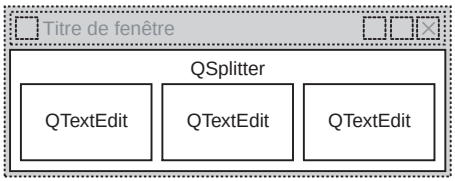


Figure 6.8
Les widgets de l'application Splitter

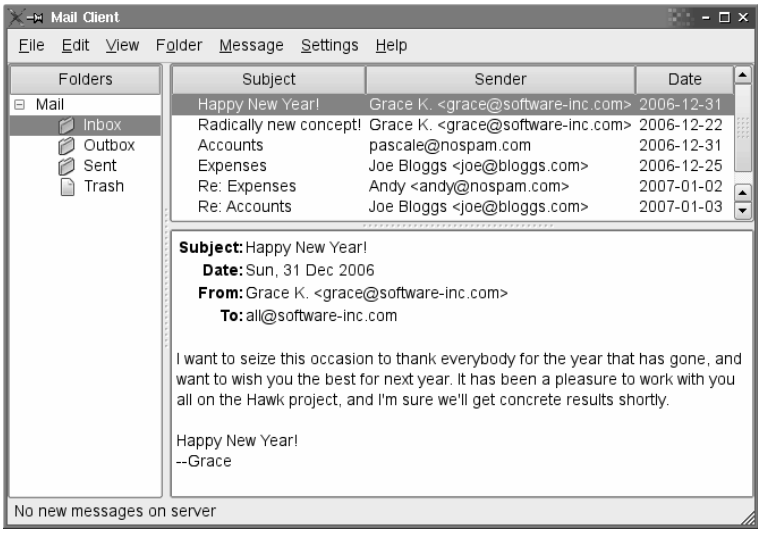


Figure 6.9
L'application Mail Client sous Mac OS X

Voici le code dans le constructeur de la sous-classe `QMainWindow` de l'application Mail Client :

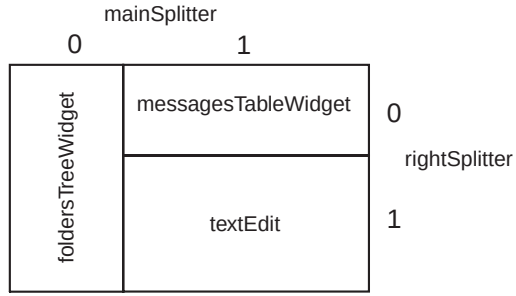
```
MailClient::MailClient()
{
    ...
    rightSplitter = new QSplitter(Qt::Vertical);
    rightSplitter->addWidget(messagesTreeWidget);
    rightSplitter->addWidget(textEdit);
    rightSplitter->setStretchFactor(1, 1);

    mainSplitter = new QSplitter(Qt::Horizontal);
    mainSplitter->addWidget(foldersTreeWidget);
    mainSplitter->addWidget(rightSplitter);
    mainSplitter->setStretchFactor(1, 1);
    setCentralWidget(mainSplitter);

    setWindowTitle(tr("Mail Client"));
    readSettings();
}
```

Après avoir créé les trois widgets que nous voulons afficher, nous créons un séparateur vertical, `rightSplitter`, et nous ajoutons les deux widgets dont nous avons besoin sur la droite.

Nous créons ensuite un séparateur horizontal, `mainSplitter`, et nous ajoutons le widget que nous voulons qu'il affiche sur la gauche. Nous créons aussi `rightSplitter` dont les widgets doivent s'afficher à droite. `mainSplitter` devient le widget central de `QMainWindow`.

Quand l'utilisateur redimensionne une fenêtre, `QSplitter` distribue normalement l'espace de sorte que les tailles relatives des widgets enfants restent les mêmes. Dans l'exemple Mail Client, nous ne souhaitons pas ce comportement ; nous voulons plutôt que `QTreeWidget` et `QTableWidget` conservent leurs dimensions et nous voulons attribuer tout espace supplémentaire à `QTextEdit` (voir Figure 6.10). Nous y parvenons grâce aux deux appels de `setStretchFactor()`. Le premier argument est l'index de base zéro du widget enfant du séparateur et le second argument est le facteur d'étirement que nous désirons définir ; la valeur par défaut est égale à 0.

Figure 6.10
Index du séparateur de l'application Mail Client



Le premier appel de `setStretchFactor()` est effectué sur `rightSplitter` et définit le widget à la position 1 (`textEdit`) pour avoir un facteur d'étirement de 1. Le deuxième appel de `setStretchFactor()` se fait sur `mainSplitter` et fixe le widget à la position 1 (`rightSplitter`) pour obtenir un facteur d'étirement de 1. Ceci garantit que tout espace supplémentaire disponible reviendra à `textEdit`.

Quand l'application est lancée, `QSplitter` attribue aux widgets enfants des tailles appropriées en fonction de leurs dimensions initiales (ou en fonction de leur taille requise si la dimension initiale n'est pas spécifiée). Nous pouvons gérer le déplacement des poignées du séparateur dans le code en appelant `QSplitter::setSizes()`. La classe `QSplitter` procure également un moyen de sauvegarder et restaurer son état la prochaine fois que l'application est exécutée. Voici la fonction `writeSettings()` qui enregistre les paramètres de Mail Client :

```
void MailClient::writeSettings()
{
   QSettings settings("Software Inc.", "Mail Client");

   settings.beginGroup("mainWindow");
   settings.setValue("size", size());
   settings.setValue("mainSplitter", mainSplitter->saveState());
   settings.setValue("rightSplitter", rightSplitter->saveState());
   settings.endGroup();
}
```

Voilà la fonction `readSettings()` correspondante :

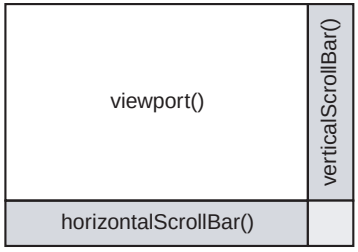
```
void MailClient::readSettings()
{
   QSettings settings("Software Inc.", "Mail Client");

   settings.beginGroup("mainWindow");
   resize(settings.value("size", QSize(480, 360)).toSize());
   mainSplitter->restoreState(
         settings.value("mainSplitter").toByteArray());
   rightSplitter->restoreState(
         settings.value("rightSplitter").toByteArray());
   settings.endGroup();
}
```

`QSplitter` est totalement pris en charge par le *Qt Designer*. Pour placer des widgets dans un séparateur, positionnez les widgets enfants plus ou moins à leurs emplacements, sélectionnez-les et cliquez sur Form > Lay Out Horizontally in Splitter ou Form > Lay Out Vertically in Splitter.

Zones déroulantes

La classe `QScrollArea` propose une fenêtre d'affichage déroulante et deux barres de défilement, comme le montre la Figure 6.11. Si vous voulez ajouter des barres de défilement à un widget, le plus simple est d'utiliser un `QScrollArea` au lieu d'instancier vos propres `QScrollBar` et d'implémenter la fonctionnalité déroulante vous-même.

Figure 6.11
Les widgets qui constituent QScrollArea



Pour se servir de `QScrollArea`, il faut appeler `setWidget()` avec le widget auquel vous souhaitez ajouter des barres de défilement. `QScrollArea` reparente automatiquement le widget pour qu'il devienne un enfant de la fenêtre d'affichage (accessible *via* `QScrollArea::viewport()`) si ce n'est pas encore le cas. Par exemple, si vous voulez des barres de défilement autour du widget `IconEditor` développé au Chapitre 5, vous avez la possibilité d'écrire ceci :

```
int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
```

```
    IconEditor *iconEditor = new IconEditor;
    iconEditor->setIconImage(QImage(":/images/mouse.png"));

    QScrollArea scrollArea;
    scrollArea.setWidget(iconEditor);
    scrollArea.viewport()->setBackgroundRole(QPalette::Dark);
    scrollArea.viewport()->setAutoFillBackground(true);
    scrollArea.setWindowTitle(QObject::tr("Icon Editor"));

    scrollArea.show();
    return app.exec();
}
```

`QScrollArea` présente le widget dans sa taille actuelle ou utilise la taille requise si le widget n'a pas encore été redimensionné. En appelant `setWidgetResizable(true)`, vous pouvez dire à `QScrollArea` de redimensionner automatiquement le widget pour profiter de tout espace supplémentaire au-delà de sa taille requise.

Par défaut, les barres de défilement ne sont affichées que lorsque la fenêtre d'affichage est plus petite que le widget enfant. Nous pouvons obliger les barres de défilement à être toujours visibles en configurant les stratégies de barre de défilement :

```
scrollArea.setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
scrollArea.setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
```

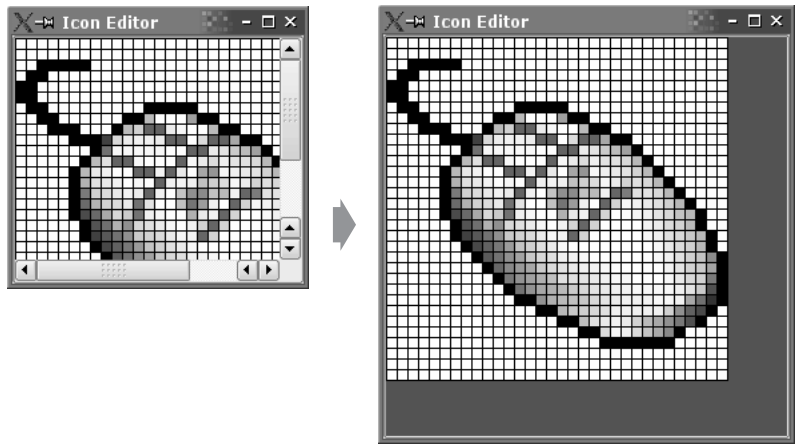


Figure 6.12
Redimensionner un QScrollArea

`QScrollArea` hérite la majorité de ses fonctionnalités de `QAbstractScrollArea`. Des classes comme `QTextEdit` et `QAbstractItemView` (la classe de base des classes d'affichage d'éléments de Qt) dérivent de `QAbstractScrollArea`, nous n'avons donc pas à les encadrer dans un `QScrollArea` pour obtenir des barres de défilement.

Widgets et barres d'outils ancrables

Les widgets ancrables sont des widgets qui peuvent être ancrés dans un `QMainWindow` ou rester flottants comme des fenêtres indépendantes. `QMainWindow` propose quatre zones d'accueil pour les widgets ancrables : une en dessous, une au-dessus, une à gauche et une à droite du widget central. Des applications telles que Microsoft Visual Studio et *Qt Linguist* utilisent énormément les fenêtres ancrables pour offrir une interface utilisateur très flexible. Dans Qt, les widgets ancrables sont des instances de `QDockWidget`.

Chaque widget ancrable possède sa propre barre de titre, même s'il est ancré (voir Figure 6.13). Les utilisateurs peuvent déplacer les fenêtres ancrables d'une zone à une autre en faisant glisser la barre de titre. Ils peuvent aussi détacher une fenêtre ancrée d'une zone et en faire une fenêtre flottante indépendante en la faisant glisser en dehors de tout point d'ancrage. Les fenêtres ancrables sont toujours affichées "au-dessus" de leur fenêtre principale lorsqu'elles sont flottantes. Les utilisateurs peuvent fermer `QDockWidget` en cliquant sur le bouton de fermeture dans la barre de titre du widget. Toute combinaison de ces fonctions peut être désactivée en appelant `QDockWidget::setFeatures()`.

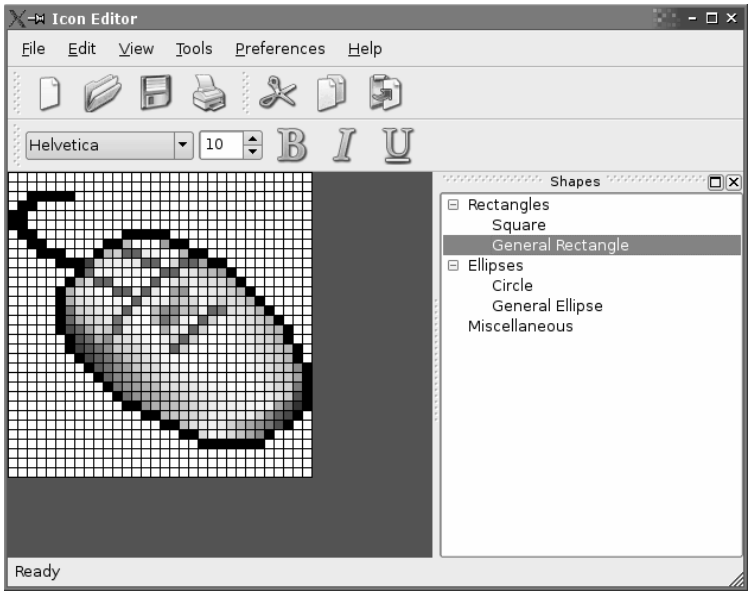


Figure 6.13
QMainWindow avec un widget ancrable

Dans les versions antérieures de Qt, les barres d'outils étaient considérées comme des widgets ancrables et partageaient les mêmes points d'ancrage. Avec Qt 4, les barres d'outils occupent leurs propres zones autour du widget central (comme illustré en Figure 6.14) et ne peuvent pas être détachées. Si une barre d'outils flottante s'avère nécessaire, nous pouvons simplement la placer dans un `QDockWindow`.

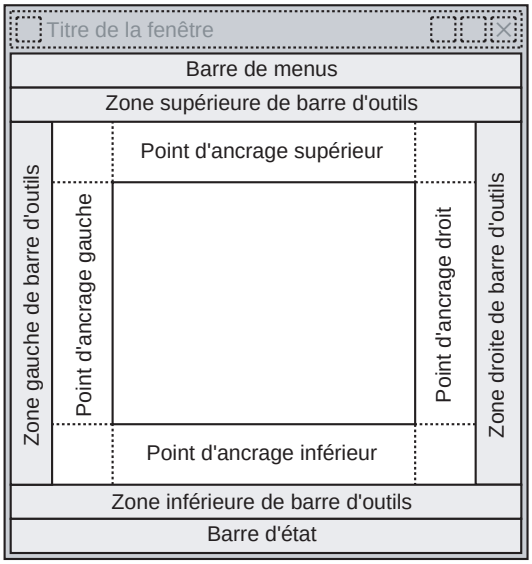


Figure 6.14
Les points d'ancrage et les zones de barres d'outils de QMainWindow

Les coins matérialisés avec des lignes pointillées peuvent appartenir à l'un des deux points d'ancrage adjacents. Par exemple, nous pourrions instaurer que le coin supérieur gauche appartient à la zone d'ancrage gauche en appelant `QMainWindow::setCorner(Qt::TopLeftCorner, Qt::LeftDockWidgetArea)`.

L'extrait de code suivant montre comment encadrer un widget existant (dans ce cas, `QTreeWidget`) dans un `QDockWidget` et comment l'insérer dans le point d'ancrage droit :

```
QDockWidget *shapesDockWidget = new QDockWidget(tr("Shapes"));
shapesDockWidget->setWidget(treeWidget);
shapesDockWidget->setAllowedAreas(Qt::LeftDockWidgetArea
                                  | Qt::RightDockWidgetArea);
addDockWidget(Qt::RightDockWidgetArea, shapesDockWidget);
```

L'appel de `setAllowedAreas()` spécifie les contraintes selon lesquelles les points d'ancrage peuvent accepter la fenêtre ancrable. Ici, nous autorisons uniquement l'utilisateur à faire glisser le widget ancrable vers les zones gauche et droite, où il y a suffisamment d'espace vertical pour qu'il s'affiche convenablement. Si aucune zone autorisée n'est explicitement spécifiée, l'utilisateur a la possibilité de faire glisser ce widget vers l'un des quatre points d'ancrage.

Voici comment créer une barre d'outils contenant un `QComboBox`, un `QSpinBox` et quelques `QToolButton` dans le constructeur d'une sous-classe de `QMainWindow` :

```
QToolBar *fontToolBar = new QToolBar(tr("Font"));
fontToolBar->addWidget(familyComboBox);
fontToolBar->addWidget(sizeSpinBox);
fontToolBar->addAction(boldAction);
fontToolBar->addAction(italicAction);
```

```
    fontToolBar->addAction(underlineAction);
    fontToolBar->setAllowedAreas(Qt::TopToolBarArea
                                 | Qt::BottomToolBarArea);
    addToolBar(fontToolBar);
```

Si nous voulons sauvegarder la position de tous les widgets ancrables et barres d'outils de manière à pouvoir les restaurer la prochaine fois que l'application sera exécutée, nous pouvons écrire un code similaire à celui utilisé pour enregistrer l'état d'un `QSplitter` à l'aide des fonctions `saveState()` et `restoreState()` de `QMainWindow` :

```
void MainWindow::writeSettings()
{
    QSettings settings("Software Inc.", "Icon Editor");

    settings.beginGroup("mainWindow");
    settings.setValue("size", size());
    settings.setValue("state", saveState());
    settings.endGroup();
}

void MainWindow::readSettings()
{
    QSettings settings("Software Inc.", "Icon Editor");

    settings.beginGroup("mainWindow");
    resize(settings.value("size").toSize());
    restoreState(settings.value("state").toByteArray());
    settings.endGroup();
}
```

Enfin, `QMainWindow` propose un menu contextuel qui répertorie toutes les fenêtres ancrables et toutes les barres d'outils, comme illustré en Figure 6.15. L'utilisateur peut fermer et restaurer ces fenêtres et masquer et restaurer des barres d'outils par le biais de ce menu.

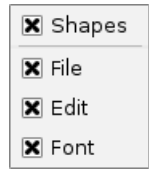


Figure 6.15
Le menu contextuel de QMainWindow

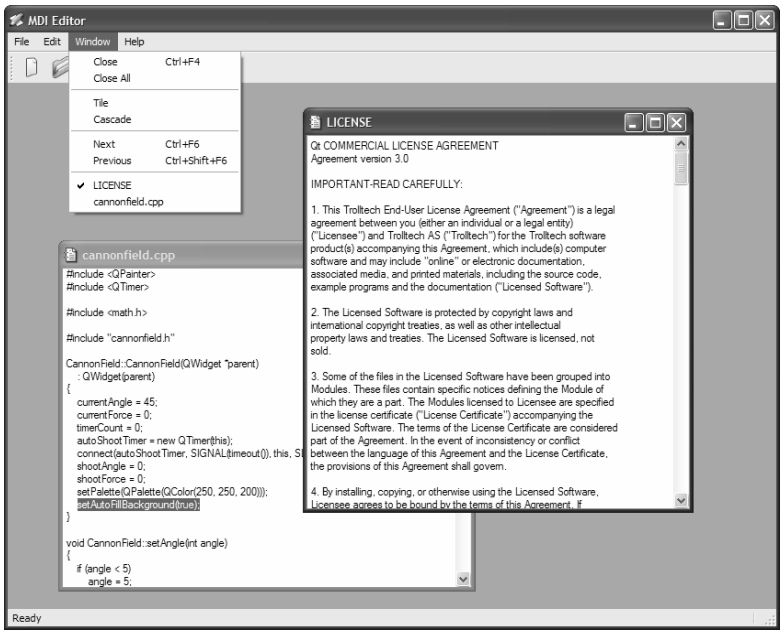
MDI (Multiple Document Interface)

Les applications qui proposent plusieurs documents dans la zone centrale de la fenêtre principale sont appelées des applications MDI (multiple document interface). Dans Qt, une application MDI est créée en utilisant la classe `QWorkspace` comme widget central et en faisant de chaque fenêtre de document un enfant de `QWorkspace`.

Les applications MDI fournissent généralement un menu Fenêtre (Window) à partir duquel vous gérez les fenêtres et les listes de fenêtres. La fenêtre active est identifiée par une coche. L'utilisateur peut activer n'importe quelle fenêtre en cliquant sur son entrée dans ce menu.

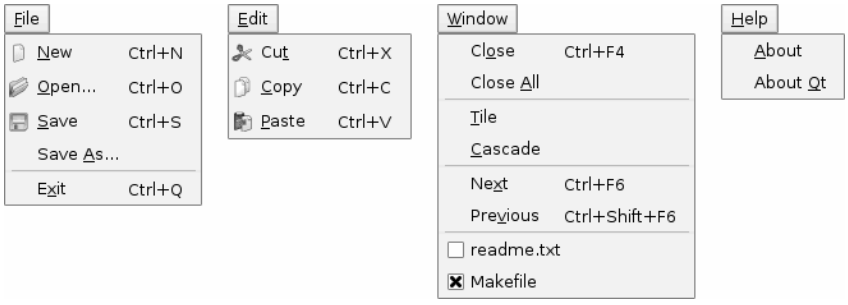
Dans cette section, nous développerons l'application MDI Editor présentée en Figure 6.16 pour vous montrer comment créer une application MDI et comment implémenter son menu Window.

Figure 6.16
L'application MDI Editor



L'application comprend deux classes : `MainWindow` et `Editor`. Le code se trouve sur le site www.pearson.fr, à la page dédiée à cet ouvrage, et vu qu'une grande partie de celui-ci est identique ou similaire à l'application Spreadsheet de la Partie I, nous ne présenterons que les parties nouvelles.

Figure 6.17
Les menus de l'application MDI Editor



Commençons par la classe `MainWindow`.

```
MainWindow::MainWindow()
{
   workspace = new QWorkspace;
   setCentralWidget(workspace);
   connect(workspace, SIGNAL(windowActivated(QWidget *)),
           this, SLOT(updateMenus()));

   createActions();
   createMenus();
   createToolBars();
   createStatusBar();

   setWindowTitle(tr("MDI Editor"));
   setWindowIcon(QPixmap(":/images/icon.png"));
}
```

Dans le constructeur `MainWindow`, nous créons un widget `QWorkspace` qui devient le widget central. Nous connectons le signal `windowActivated()` de `QWorkspace` au slot que nous voulons utiliser pour conserver le menu Window à jour.

```
void MainWindow::newFile()
{
   Editor *editor = createEditor();
   editor->newFile();
   editor->show();
}
```

Le slot `newFile()` correspond à l'option File > New. Il dépend de la fonction privée `createEditor()` pour créer un widget enfant `Editor`.

```
Editor *MainWindow::createEditor()
{
   Editor *editor = new Editor;
   connect(editor, SIGNAL(copyAvailable(bool)),
           cutAction, SLOT(setEnabled(bool)));
   connect(editor, SIGNAL(copyAvailable(bool)),
           copyAction, SLOT(setEnabled(bool)));

   workspace->addWindow(editor);
   windowMenu->addAction(editor->windowMenuAction());
   windowActionGroup->addAction(editor->windowMenuAction());

   return editor;
}
```

La fonction `createEditor()` crée un widget `Editor` et établit deux connexions signal-slot. Ces connexions garantissent que Edit > Cut et Edit > Copy sont activés ou désactivés selon que du texte est sélectionné ou non.

Avec MDI, il est possible que plusieurs widgets `Editor` soient utilisés. C'est un souci parce que nous ne voulons répondre qu'au signal `copyAvailable(bool)` de la fenêtre `Editor` active et pas aux autres. Cependant, ces signaux ne peuvent être émis que par la fenêtre active, ce n'est donc pas un problème en pratique.

Lorsque nous avons configuré `Editor`, nous avons ajouté un `QAction` représentant la fenêtre au menu Window. L'action est fournie par la classe `Editor` que nous étudierons plus loin. Nous ajoutons également l'action à un objet `QActionGroup`. Avec `QActionGroup`, nous sommes sûrs qu'un seul élément du menu `Window` est coché à la fois.

```
void MainWindow::open()
{
   Editor *editor = createEditor();
   if (editor->open()) {
       editor->show();
   } else {
       editor->close();
   }
}
```

La fonction `open()` correspond à File > Open. Elle crée un `Editor` pour le nouveau document et appelle `open()` sur ce dernier. Il est préférable d'implémenter des opérations de fichier dans la classe `Editor` que dans la classe `MainWindow`, parce que chaque `Editor` a besoin d'assurer le suivi de son propre état indépendant.

Si `open()` échoue, nous fermons simplement l'éditeur parce que l'utilisateur aura déjà été informé de l'erreur. Nous n'avons pas à supprimer explicitement l'objet `Editor` nous-mêmes ; c'est fait automatiquement par `Editor` par le biais de l'attribut `Qt::WA_DeleteOnClose`, qui est défini dans le constructeur de `Editor`.

```
void MainWindow::save()
{
   if (activeEditor())
       activeEditor()->save();
}
```

Le slot `save()` invoque `Editor::save()` sur l'éditeur actif, s'il y en a un. Une fois encore, le code qui accomplit le véritable travail se situe dans la classe `Editor`.

```
Editor *MainWindow::activeEditor()
{
   return qobject_cast<Editor *>(workspace->activeWindow());
}
```

La fonction privée `activeEditor()` retourne la fenêtre enfant active sous la forme d'un pointeur de `Editor`, ou d'un pointeur nul s'il n'y en a pas.

```
void MainWindow::cut()
{
   if (activeEditor())
       activeEditor()->cut();
}
```

Le slot `cut()` invoque `Editor::cut()` sur l'éditeur actif. Nous ne montrons pas les slots `copy()` et `paste()` puisqu'ils suivent le même modèle.

```
void MainWindow::updateMenus()
{
   bool hasEditor = (activeEditor() != 0);
   bool hasSelection = activeEditor()
                       && activeEditor()->textCursor().hasSelection();

   saveAction->setEnabled(hasEditor);
   saveAsAction->setEnabled(hasEditor);
   pasteAction->setEnabled(hasEditor);
   cutAction->setEnabled(hasSelection);
   copyAction->setEnabled(hasSelection);
   closeAction->setEnabled(hasEditor);
   closeAllAction->setEnabled(hasEditor);
   tileAction->setEnabled(hasEditor);
   cascadeAction->setEnabled(hasEditor);
   nextAction->setEnabled(hasEditor);
   previousAction->setEnabled(hasEditor);
   separatorAction->setVisible(hasEditor);

   if (activeEditor())
       activeEditor()->windowMenuAction()->setChecked(true);
}
```

Le slot `updateMenus()` est invoqué dès qu'une fenêtre est activée (et quand la dernière fenêtre est fermée) pour mettre à jour le système de menus, en raison de la connexion signal-slot dans le constructeur de `MainWindow`.

La plupart des options de menu ne sont intéressantes que si une fenêtre est active, nous les désactivons donc quand ce n'est pas le cas. Nous terminons en appelant `setChecked()` sur `QAction` représentant la fenêtre active. Grâce à `QActionGroup`, nous n'avons pas besoin de décocher explicitement la fenêtre active précédente.

```
void MainWindow::createMenus()
{
   ...
   windowMenu = menuBar()->addMenu(tr("&Window"));
   windowMenu->addAction(closeAction);
   windowMenu->addAction(closeAllAction);
   windowMenu->addSeparator();
   windowMenu->addAction(tileAction);
   windowMenu->addAction(cascadeAction);
   windowMenu->addSeparator();
   windowMenu->addAction(nextAction);
   windowMenu->addAction(previousAction);
   windowMenu->addAction(separatorAction);
   ...
}
```

La fonction privée `createMenus()` introduit des actions dans le menu Window. Les actions sont toutes typiques de tels menus et sont implémentées facilement à l'aide des slots `closeActiveWindow()`, `closeAllWindows()`, `tile()` et `cascade()` de `QWorkspace`. A chaque fois que l'utilisateur ouvre une nouvelle fenêtre, elle est ajoutée à la liste d'actions du menu Window. (Ceci est effectué dans la fonction `createEditor()` étudiée précédemment.) Dès que l'utilisateur ferme une fenêtre d'éditeur, son action dans le menu Window est supprimée (étant donné que l'action appartient à la fenêtre d'éditeur), et elle disparaît de ce menu.

```
void MainWindow::closeEvent(QCloseEvent *event)
{
   workspace->closeAllWindows();
   if (activeEditor()) {
      event->ignore();
   } else {
      event->accept();
   }
}
```

La fonction `closeEvent()` est réimplémentée pour fermer toutes les fenêtres enfants, chaque enfant reçoit donc un événement `close`. Si l'un des widgets enfants "ignore" cet événement (parce que l'utilisateur a annulé une boîte de message "modifications non enregistrées"), nous ignorons l'événement `close` pour `MainWindow` ; sinon, nous l'acceptons, ce qui a pour conséquence de fermer la fenêtre complète. Si nous n'avions pas réimplémenté `closeEvent()` dans `MainWindow`, l'utilisateur n'aurait pas eu la possibilité de sauvegarder des modifications non enregistrées.

Nous avons terminé notre analyse de `MainWindow`, nous pouvons donc passer à l'implémentation d'`Editor`. La classe `Editor` représente une fenêtre enfant. Elle hérite de `QTextEdit` qui propose une fonctionnalité de modification de texte. Tout comme n'importe quel widget Qt peut être employé comme une fenêtre autonome, tout widget Qt peut être utilisé comme une fenêtre enfant dans un espace de travail MDI.

Voici la définition de classe :

```
class Editor : public QTextEdit
{
   Q_OBJECT

public:
   Editor(QWidget *parent = 0);

   void newFile();
   bool open();
   bool openFile(const QString &fileName);
   bool save();
   bool saveAs();
   QSize sizeHint() const;
   QAction *windowMenuAction() const { return action; }
```

```
protected:
   void closeEvent(QCloseEvent *event);

private slots:
   void documentWasModified();

private:
   bool okToContinue();
   bool saveFile(const QString &fileName);
   void setCurrentFile(const QString &fileName);
   bool readFile(const QString &fileName);
   bool writeFile(const QString &fileName);
   QString strippedName(const QString &fullFileName);

   QString curFile;
   bool isUntitled;
   QString fileFilters;
   QAction *action;
};
```

Quatre des fonctions privées qui se trouvaient dans la classe `MainWindow` de l'application Spreadsheet sont également présentes dans la classe `Editor` : `okToContinue()`, `saveFile()`, `setCurrentFile()` et `strippedName()`.

```
Editor::Editor(QWidget *parent)
   : QTextEdit(parent)
{
   action = new QAction(this);
   action->setCheckable(true);
   connect(action, SIGNAL(triggered()), this, SLOT(show()));
   connect(action, SIGNAL(triggered()), this, SLOT(setFocus()));

   isUntitled = true;
   fileFilters = tr("Text files (*.txt)\n"
                    "All files (*)");

   connect(document(), SIGNAL(contentsChanged()),
           this, SLOT(documentWasModified()));

   setWindowIcon(QPixmap(":/images/document.png"));
   setAttribute(Qt::WA_DeleteOnClose);
}
```

Nous créons d'abord un `QAction` représentant l'éditeur dans le menu Window de l'application et nous connectons cette action aux slots `show()` et `setFocus()`.

Etant donné que nous autorisons les utilisateurs à créer autant de fenêtres d'éditeurs qu'ils le souhaitent, nous devons prendre certaines dispositions concernant leur dénomination, dans le but de faire une distinction avant le premier enregistrement. Un moyen courant de gérer cette situation est d'attribuer des noms qui incluent un chiffre (par exemple, document1.txt). Nous utilisons la variable `isUntitled` pour faire une distinction entre les noms fournis par l'utilisateur et les noms créés par programme.

Nous connectons le signal `contentsChanged()` du document au slot privé `documentWasModified()`. Ce slot appelle simplement `setWindowModified(true)`.

Enfin, nous définissons l'attribut `Qt::WA_DeleteOnClose` pour éviter toute fuite de mémoire quand l'utilisateur ferme une fenêtre `Editor`.

Après le constructeur, il est logique d'appeler `newFile()` ou `open()`.

```
void Editor::newFile()
{
   static int documentNumber = 1;

   curFile = tr("document%1.txt").arg(documentNumber);
   setWindowTitle(curFile + "[*]");
   action->setText(curFile);
   isUntitled = true;
   ++documentNumber;
}
```

La fonction `newFile()` génère un nom au format document1.txt pour le nouveau document. Ce code est placé dans `newFile()` plutôt que dans le constructeur, parce qu'il n'y a aucun intérêt à numéroter quand nous invoquons `open()` pour ouvrir un document existant dans un `Editor` nouvellement créé. `documentNumber` étant déclaré statique, il est partagé par toutes les instances d'`Editor`.

Le symbole "[*]" dans le titre de la fenêtre réserve l'emplacement de l'astérisque qui doit apparaître quand le fichier contient des modifications non sauvegardées sur des plates-formes autres que Mac OS X. Nous avons parlé de ce symbole dans le Chapitre 3.

```
bool Editor::open()
{
   QString fileName =
           QFileDialog::getOpenFileName(this, tr("Open"), ".",
                                        fileFilters);
   if (fileName.isEmpty())
       return false;

   return openFile(fileName);
}
```

La fonction `open()` essaie d'ouvrir un fichier existant avec `openFile()`.

```
bool Editor::save()
{
   if (isUntitled) {
      return saveAs();
   } else {
      return saveFile(curFile);
   }
}
```

La fonction `save()` s'appuie sur la variable `isUntitled` pour déterminer si elle doit appeler `saveFile()` ou `saveAs()`.

```
void Editor::closeEvent(QCloseEvent *event)

{
   if (okToContinue()) {
      event->accept();
   } else {
      event->ignore();
   }
}
```

La fonction `closeEvent()` est réimplémentée pour permettre à l'utilisateur de sauvegarder des modifications non enregistrées. La logique est codée dans la fonction `okToContinue()` qui ouvre une boîte de message demandant, "Voulez-vous enregistrer vos modifications ?" Si `okToContinue()` retourne `true`, nous acceptons l'événement `close` ; sinon, nous "l'ignorons" et la fenêtre n'en sera pas affectée.

```
void Editor::setCurrentFile(const QString &fileName)
{
   curFile = fileName;
   isUntitled = false;
   action->setText(strippedName(curFile));

   document()->setModified(false);
   setWindowTitle(strippedName(curFile) + "[*]");
   setWindowModified(false);
}
```

La fonction `setCurrentFile()` est appelée dans `openFile()` et `saveFile()` pour mettre à jour les variables `curFile` et `isUntitled`, pour définir le titre de la fenêtre et le texte de l'action et pour configurer l'indicateur "modi?ed" du document en `false`. Dès que l'utilisateur modifie le texte dans l'éditeur, le `QTextDocument` sous-jacent émet le signal `contentsChanged()` et définit son indicateur "modified" interne en `true`.

```
QSize Editor::sizeHint() const
{
   return QSize(72 * fontMetrics().width('x'),
                25 * fontMetrics().lineSpacing());
}
```

La fonction `sizeHint()` retourne une taille en fonction de la largeur de la lettre "x" et de la hauteur de la ligne de texte. `QWorkspace` se sert de la taille requise pour attribuer une dimension initiale à la fenêtre.

Voici le fichier `main.cpp` de l'application MDI Editor :

```
#include <QApplication>

#include "mainwindow.h"
```

```
int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   QStringList args = app.arguments();

   MainWindow mainWin;
   if (args.count() > 1) {
      for (int i = 1; i < args.count(); ++i)
         mainWin.openFile(args[i]);
   } else {
      mainWin.newFile();
   }

   mainWin.show();
   return app.exec();
}
```

Si l'utilisateur spécifie des fichiers dans la ligne de commande, nous tentons de les charger. Sinon, nous démarrons avec un document vide. Les options de ligne de commande spécifiques à Qt, comme `-style` et `-font`, sont automatiquement supprimées de la liste d'arguments par le constructeur `QApplication`. Donc si nous écrivons :

```
mdieditor -style motif readme.txt
```

dans la ligne de commande, `QApplication::arguments()` retourne un `QStringList` contenant deux éléments (`"mdieditor"` et `"readme.txt"`) et l'application MDI Editor démarre avec le document `readme.txt`.

MDI est un moyen de gérer simultanément plusieurs documents. Sous Mac OS X, la meilleure approche consiste à utiliser plusieurs fenêtres de haut niveau. Cette technique est traitée dans la section "Documents multiples" du Chapitre 3.

7

Traitement des événements

Au sommaire de ce chapitre

- ✔ Réimplémenter les gestionnaires d'événements
- ✔ Installer des filtres d'événements
- ✔ Rester réactif pendant un traitement intensif

Les événements sont déclenchés par le système de fenêtrage ou par Qt en réponse à diverses circonstances. Quand l'utilisateur enfonce ou relâche une touche ou un bouton de la souris, un événement `key` ou `mouse` est déclenché ; lorsqu'une fenêtre s'affiche pour la première fois, un événement `paint` est généré pour informer la fenêtre nouvellement affichée qu'elle doit se redessiner. La plupart des événements sont déclenchés en réponse à des actions utilisateur, mais certains, comme les événements `timer`, sont déclenchés indépendamment par le système.

Quand nous programmons avec Qt, nous avons rarement besoin de penser aux événements, parce que les widgets Qt émettent des signaux lorsque quelque chose de significatif se produit. Les événements deviennent utiles quand nous écrivons nos propres widgets personnalisés ou quand nous voulons modifier le comportement des widgets Qt existants.

Il ne faut pas confondre événements et signaux. En règle générale, les signaux s'avèrent utiles lors de *l'emploi* d'un widget, alors que les événements présentent une utilité au moment de *l'implémentation* d'un widget. Par exemple, quand nous utilisons `QPushButton`, nous nous intéressons plus à son signal `clicked()` qu'aux événements `mouse` ou `key` de bas niveau qui provoquent l'émission du signal. Cependant, si nous implémentons une classe telle que `QPushButton`, nous devons écrire un code qui gérera les événements `mouse` et `key` et qui émettra le signal `clicked()` si nécessaire.

Réimplémenter les gestionnaires d'événements

Dans Qt, un événement est un objet qui hérite de `QEvent`. Qt gère plus d'une centaine de types d'événements, chacun d'eux étant identifié par une valeur d'énumération. Par exemple, `QEvent::type()` retourne `QEvent::MouseButtonPress` pour les événements "bouton souris enfoncé".

De nombreux types d'événements exigent plus d'informations que ce qui peut être stocké dans un objet `QEvent` ordinaire ; par exemple, les événements "bouton souris enfoncé" doivent stocker quel bouton de la souris a déclenché l'événement et l'endroit où le pointeur de la souris se trouvait quand l'événement s'est déclenché. Ces informations supplémentaires sont conservées dans des sous-classes `QEvent` spéciales, comme `QMouseEvent`.

Les événements sont notifiés aux objets par le biais de leur fonction `event()`, héritée de `QObject`. L'implémentation de `event()` dans `QWidget` transmet les types les plus courants d'événements à des gestionnaires d'événements spécifiques, tels que `mousePressEvent()`, `keyPressEvent()` et `paintEvent()`.

Nous avons déjà étudié plusieurs gestionnaires d'événements lorsque nous avons implémenté `MainWindow`, `IconEditor` et `Plotter` dans les chapitres précédents. Il existe beaucoup d'autres types d'événements répertoriés dans la documentation de référence de `QEvent`, et il est aussi possible de créer des types d'événements personnalisés et d'envoyer les événements soi-même. Dans notre cas, nous analyserons deux types courants d'événements qui méritent davantage d'explications : les événements `key` et `timer`.

Les événements `key` sont gérés en réimplémentant `keyPressEvent()` et `keyReleaseEvent()`. Le widget `Plotter` réimplémente `keyPressEvent()`. Normalement, nous ne devons réimplémenter que `keyPressEvent()` puisque les seules touches pour lesquelles il faut contrôler qu'elles ont été relâchées sont les touches de modification Ctrl, Maj et Alt, et vous pouvez contrôler leur état dans un `keyPressEvent()` en utilisant `QKeyEvent::modifiers()`.

Par exemple, si nous implémentions un widget `CodeEditor`, voici le code de sa fonction `keyPressEvent()` qui devra interpréter différemment `Home` et `Ctrl+Home` :

```
void CodeEditor::keyPressEvent(QKeyEvent *event)
{
   switch (event->key()) {
   case Qt::Key_Home:
      if (event->modifiers() & Qt::ControlModifier) {
          goToBeginningOfDocument();
      } else {
          goToBeginningOfLine();
      }
      break;
   case Qt::Key_End:
      ...
   default:
      QWidget::keyPressEvent(event);
   }
}
```

Les touches de tabulation et de tabulation arrière (Maj+Tab) sont des cas particuliers. Elles sont gérées par `QWidget::event()` avant l'appel de `keyPressEvent()`, avec la consigne de transmettre le focus au widget suivant ou précédent dans l'ordre de la chaîne de focus. Ce comportement correspond habituellement à ce que nous recherchons, mais dans un widget `CodeEditor`, nous préférerions que la touche Tab produise le décalage d'une ligne par rapport à la marge. Voici comment `event()` pourrait être réimplémenté :

```
bool CodeEditor::event(QEvent *event)
{
   if (event->type() == QEvent::KeyPress) {
       QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
       if (keyEvent->key() == Qt::Key_Tab) {
           insertAtCurrentPosition('\t');
           return true;
      }
   }
   return QWidget::event(event);
}
```

Si l'événement est lié à une touche sur laquelle l'utilisateur a appuyé, nous convertissons l'objet `QEvent` en `QKeyEvent` et nous vérifions quelle touche a été pressée. S'il s'agit de la touche Tab, nous effectuons un traitement et nous retournons `true` pour informer Qt que nous avons géré l'événement. Si nous avions retourné `false`, Qt transmettrait l'événement au widget parent.

Une approche plus intelligente pour implémenter les combinaisons de touches consiste à se servir de `QAction`. Par exemple, si `goToBeginningOfLine()` et `goToBeginningOf-Document()` sont des slots publics dans le widget `CodeEditor`, et si `CodeEditor` fait

office de widget central dans une classe `MainWindow`, nous pourrions ajouter des combinaisons de touches avec le code suivant :

```
MainWindow::MainWindow()
{
   editor = new CodeEditor;
   setCentralWidget(editor);

   goToBeginningOfLineAction =
         new QAction(tr("Go to Beginning of Line"), this);
   goToBeginningOfLineAction->setShortcut(tr("Home"));
   connect(goToBeginningOfLineAction, SIGNAL(activated()),
         editor, SLOT(goToBeginningOfLine()));

   goToBeginningOfDocumentAction =
         new QAction(tr("Go to Beginning of Document"), this);
   goToBeginningOfDocumentAction->setShortcut(tr("Ctrl+Home"));
   connect(goToBeginningOfDocumentAction, SIGNAL(activated()),
         editor, SLOT(goToBeginningOfDocument()));
   ...
}
```

Cela permet d'ajouter facilement des commandes à un menu ou une barre d'outils, comme nous l'avons vu dans le Chapitre 3. Si les commandes n'apparaissent pas dans l'interface utilisateur, les objets `QAction` pourraient être remplacés par un objet `QShortcut`, la classe employée par `QAction` en interne pour prendre en charge les combinaisons de touches.

Par défaut, les combinaisons de touches définies à l'aide de `QAction` ou `QShortcut` sur un widget sont activées dès que la fenêtre contenant le widget est active. Vous pouvez modifier ce comportement grâce à `QAction::setShortcutContext()` ou `QShortcut::setContext()`.

L'autre type courant d'événement est l'événement `timer`. Alors que la plupart des autres types d'événements se déclenchent suite à une action utilisateur, les événements `timer` permettent aux applications d'effectuer un traitement à intervalles réguliers. Les événements `timer` peuvent être utilisés pour implémenter des curseurs clignotants et d'autres animations, ou simplement pour réactualiser l'affichage.

Pour analyser les événements `timer`, nous implémenterons un widget `Ticker` illustré en Figure 7.1. Ce widget présente une bannière qui défile d'un pixel vers la gauche toutes les 30 millisecondes. Si le widget est plus large que le texte, le texte est répété autant de fois que nécessaire pour remplir toute la largeur du widget.

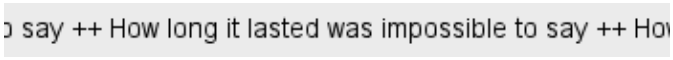


Figure 7.1
Le widget Ticker

Voici le fichier d'en-tête :

```
#ifndef TICKER_H
#define TICKER_H
```

```
#include <QWidget>

class Ticker : public QWidget
{
   Q_OBJECT
   Q_PROPERTY(QString text READ text WRITE setText)

public:
   Ticker(QWidget *parent = 0);

   void setText(const QString &newText);
   QString text() const { return myText; }
   QSize sizeHint() const;

protected:
   void paintEvent(QPaintEvent *event);
   void timerEvent(QTimerEvent *event);
   void showEvent(QShowEvent *event);
   void hideEvent(QHideEvent *event);

private:
   QString myText;
   int offset;
   int myTimerId;
};

#endif
```

Nous réimplémentons quatre gestionnaires d'événements dans `Ticker`, dont trois que nous avons déjà vus auparavant : `timerEvent()`, `showEvent()` et `hideEvent()`.

Analysons à présent l'implémentation :

```
#include <QtGui>

#include "ticker.h"

Ticker::Ticker(QWidget *parent)
   : QWidget(parent)
{
   offset = 0;
   myTimerId = 0;
}
```

Le constructeur initialise la variable `offset` à 0. Les coordonnées *x* auxquelles le texte est dessiné sont calculées à partir de la valeur `offset`. Les ID du timer sont toujours différents de zéro, nous utilisons donc 0 pour indiquer qu'aucun timer n'a été démarré.

```
void Ticker::setText(const QString &newText)
{
   myText = newText;
   update();
   updateGeometry();
}
```

La fonction `setText()` détermine le texte à afficher. Elle invoque `update()` pour demander le rafraîchissement de l'affichage et `updateGeometry()` pour informer tout gestionnaire de disposition responsable du widget `Ticker` d'un changement de taille requise.

```
QSize Ticker::sizeHint() const
{
   return fontMetrics().size(0, text());
}
```

La fonction `sizeHint()` retourne l'espace requis par le texte comme étant la taille idéale du widget. `QWidget::fontMetrics()` renvoie un objet `QFontMetrics` qui peut être interrogé pour obtenir des informations liées à la police du widget. Dans ce cas, nous demandons la taille exigée par le texte. (Puisque le premier argument de `QFontMetrics::size()` est un indicateur qui n'est pas nécessaire pour les chaînes simples, nous transmettons simplement 0.)

```
void Ticker::paintEvent(QPaintEvent * /* event */)
{
   QPainter painter(this);

   int textWidth = fontMetrics().width(text());
   if (textWidth < 1)
      return;
   int x = -offset;
   while (x < width()) {
      painter.drawText(x, 0, textWidth, height(),
                       Qt::AlignLeft | Qt::AlignVCenter, text());
      x += textWidth;
   }
}
```

La fonction `paintEvent()` dessine le texte avec `QPainter::drawText()`. Elle détermine la quantité d'espace horizontal exigé par le texte à l'aide de `fontMetrics()`, puis dessine le texte autant de fois que nécessaire pour remplir toute la largeur du widget, en tenant compte du décalage `offset`.

```
void Ticker::showEvent(QShowEvent * /* event */)
{
   myTimerId = startTimer(30);
}
```

La fonction `showEvent()` lance un timer. L'appel de `QObject::startTimer()` retourne un ID, qui peut être utilisé ultérieurement pour identifier le timer. `QObject` prend en charge plusieurs timers indépendants, chacun possédant son propre intervalle de temps. Après l'appel de `startTimer()`, Qt déclenche un événement `timer` environ toutes les 30 millisecondes ; la précision dépend du système d'exploitation sous-jacent.

Nous aurions pu appeler `startTimer()` dans le constructeur de `Ticker`, mais nous économisons des ressources en ne déclenchant des événements `timer` que lorsque le widget est visible.

```
void Ticker::timerEvent(QTimerEvent *event)
```

```
{
   if (event->timerId() == myTimerId) {
      ++offset;
      if (offset >= fontMetrics().width(text()))
         offset = 0;
      scroll(-1, 0);
   } else {
      QWidget::timerEvent(event);
   }
}
```

La fonction `timerEvent()` est invoquée à intervalles réguliers par le système. Elle incrémente `offset` de 1 pour simuler un mouvement, encadrant la largeur du texte. Puis elle fait défiler le contenu du widget d'un pixel vers la gauche grâce à `QWidget::scroll()`. Nous aurions pu simplement appeler `update()` au lieu de `scroll()`, mais `scroll()` est plus efficace, parce qu'elle déplace simplement les pixels existants à l'écran et ne déclenche un événement `paint` que pour la zone nouvellement affichée du widget (une bande d'un pixel de large dans ce cas).

Si l'événement `timer` ne correspond pas au timer qui nous intéresse, nous le transmettons à notre classe de base.

```
void Ticker::hideEvent(QHideEvent * /* event */)
{
   killTimer(myTimerId);
}
```

La fonction `hideEvent()` invoque `QObject::killTimer()` pour arrêter le timer.

Les événements `timer` sont de bas niveau, et si nous avons besoin de plusieurs timers, il peut être fastidieux d'assurer le suivi de tous les ID de timer. Dans de telles situations, il est généralement plus facile de créer un objet `QTimer` pour chaque timer. `QTimer` émet le signal `timeout()` à chaque intervalle de temps. `QTimer` propose aussi une interface pratique pour les timers à usage unique (les timers qui ne chronomètrent qu'une seule fois).

Installer des filtres d'événements

L'une des fonctionnalités vraiment puissante du modèle d'événement de Qt est qu'une instance de `QObject` peut être configurée de manière à contrôler les événements d'une autre instance de `QObject` avant même que cette dernière ne les détecte.

Supposons que nous avons un widget `CustomerInfoDialog` composé de plusieurs `QLineEdit` et que nous voulons utiliser la barre d'espace pour activer le prochain `QLineEdit`. Ce comportement inhabituel peut se révéler approprié pour une application interne à laquelle les utilisateurs sont formés. Une solution simple consiste à dériver `QLineEdit` et à réimplémenter `keyPressEvent()` pour appeler `focusNextChild()`, comme dans le code suivant :

```
void MyLineEdit::keyPressEvent(QKeyEvent *event)
{
```

```
    if (event->key() == Qt::Key_Space) {
       focusNextChild();
    } else {
       QLineEdit::keyPressEvent(event);
    }
}
```

Cette approche présente un inconvénient de taille : si nous utilisons plusieurs types de widgets dans le formulaire (par exemple, `QComboBoxes` et `QSpinBoxes`), nous devons également les dériver pour qu'ils affichent le même comportement. Il existe une meilleure solution : `CustomerInfoDialog` contrôle les événements "bouton souris enfoncé" de ses widgets enfants et implémente le comportement nécessaire dans le code de contrôle. Pour ce faire, vous utiliserez des filtres d'événements. Définir un filtre d'événement implique deux étapes :

1. enregistrer l'objet contrôleur avec l'objet cible en appelant `installEventFilter()` sur la cible ;
2. gérer les événements de l'objet cible dans la fonction `eventFilter()` de l'objet contrôleur.

Le code du constructeur de `CustomerInfoDialog` constitue un bon endroit pour enregistrer l'objet contrôleur :

```
CustomerInfoDialog::CustomerInfoDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    firstNameEdit->installEventFilter(this);
    lastNameEdit->installEventFilter(this);
    cityEdit->installEventFilter(this);
    phoneNumberEdit->installEventFilter(this);
}
```

Dès que le filtre d'événement est enregistré, les événements qui sont envoyés aux widgets `firstNameEdit`, `lastNameEdit`, `cityEdit` et `phoneNumberEdit` sont d'abord transmis à la fonction `eventFilter()` de `CustomerInfoDialog` avant d'être envoyés vers la destination prévue.

Voici la fonction `eventFilter()` qui reçoit les événements :

```
bool CustomerInfoDialog::eventFilter(QObject *target, QEvent *event)
{
    if (target == firstNameEdit || target == lastNameEdit
          || target == cityEdit || target == phoneNumberEdit) {
       if (event->type() == QEvent::KeyPress) {
          QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
          if (keyEvent->key() == Qt::Key_Space) {
             focusNextChild();
             return true;
          }
       }
    }
    return QDialog::eventFilter(target, event);
}
```

Nous vérifions tout d'abord que le widget cible est un des `QLineEdit`. Si l'événement est lié à l'enfoncement d'une touche, nous le convertissons en `QKeyEvent` et nous vérifions quelle touche a été pressée. Si la touche enfoncée correspondait à la barre d'espace, nous invoquons `focusNextChild()` pour activer le prochain widget dans la chaîne de focus, et nous retournons `true` pour dire à Qt que nous avons géré l'événement. Si nous avions renvoyé `false`, Qt aurait envoyé l'événement à sa cible prévue, ce qui aurait introduit un espace parasite dans `QLineEdit`.

Si le widget cible n'est pas un `QLineEdit`, ou si l'événement ne résulte pas de l'enfoncement de la barre d'espace, nous passons le contrôle à l'implémentation de `eventFilter()` de la classe de base. Le widget cible aurait aussi pu être un widget que la classe de base, `QDialog`, est en train de contrôler. (Dans Qt 4.1, ce n'est pas le cas pour `QDialog`. Cependant, d'autres classes de widgets Qt, comme `QScrollArea`, surveillent certains de leurs widgets enfants pour diverses raisons.)

Qt propose cinq niveaux auxquels des événements peuvent être traités et filtrés :

1. **Nous pouvons réimplémenter un gestionnaire d'événements spécifique.**

   Réimplémenter des gestionnaires d'événements comme `mousePressEvent()`, `keyPressEvent()` et `paintEvent()` est de loin le moyen le plus commun de traiter des événements. Nous en avons déjà vu de nombreux exemples.

2. **Nous pouvons réimplémenter `QObject::event()`.**

   En réimplémentant la fonction `event()`, nous avons la possibilité de traiter des événements avant qu'ils n'atteignent les gestionnaires d'événements spécifiques. Cette approche est surtout employée pour redéfinir la signification par défaut de la touche Tab, comme expliqué précédemment. Elle est aussi utilisée pour gérer des types rares d'événements pour lesquels il n'existe aucun gestionnaire d'événements spécifique (par exemple, `QEvent::HoverEnter`). Quand nous réimplémentons `event()`, nous devons appeler la fonction `event()` de la classe de base pour gérer les cas que nous ne gérons pas explicitement.

3. **Nous pouvons installer un filtre d'événement sur un seul `QObject`.**

   Lorsqu'un objet a été enregistré avec `installEventFilter()`, tous les événements pour l'objet cible sont d'abord envoyés à la fonction `eventFilter()` de l'objet contrôleur. Si plusieurs filtres d'événements sont installés sur le même objet, les filtres sont activés à tour de rôle, du plus récemment installé au premier installé.

4. **Nous pouvons installer un filtre d'événement sur l'objet `QApplication`.**

   Lorsqu'un filtre d'événement a été enregistré pour `qApp` (l'unique objet de `QApplication`), chaque événement de chaque objet de l'application est envoyé à la fonction `eventFilter()` avant d'être transmis à un autre filtre d'événement. Cette technique est très utile pour le débogage. Elle peut aussi être employée pour gérer des événements `mouse` transmis aux widgets désactivés que `QApplication` ignore normalement.

5. **Nous pouvons dériver `QApplication` et réimplémenter `notify()`.**

   Qt appelle `QApplication::notify()` pour envoyer un événement. Réimplémenter cette fonction est le seul moyen de récupérer tous les événements avant qu'un filtre d'événement quelconque n'ait l'opportunité de les analyser. Les filtres d'événements sont généralement plus pratiques, parce que le nombre de filtres concomitants n'est pas limité alors qu'il ne peut y avoir qu'une seule fonction `notify()`.

De nombreux types d'événements, dont les événements `mouse` et `key`, peuvent se propager. Si l'événement n'a pas été géré lors de son trajet vers son objet cible ou par l'objet cible lui-même, tout le traitement de l'événement est répété, mais cette fois-ci avec comme cible le parent de l'objet cible initial. Ce processus se poursuit, en passant d'un parent à l'autre, jusqu'à ce que l'événement soit géré ou que l'objet de niveau supérieur soit atteint.

La Figure 7.2 vous montre comment un événement "bouton souris enfoncé" est transmis d'un enfant vers un parent dans une boîte de dialogue. Quand l'utilisateur appuie sur une touche, l'événement est d'abord envoyé au widget actif, dans ce cas le `QCheckBox` en bas à droite. Si le `QCheckBox` ne gère pas l'événement, Qt l'envoie au `QGroupBox` et enfin à l'objet `QDialog`.

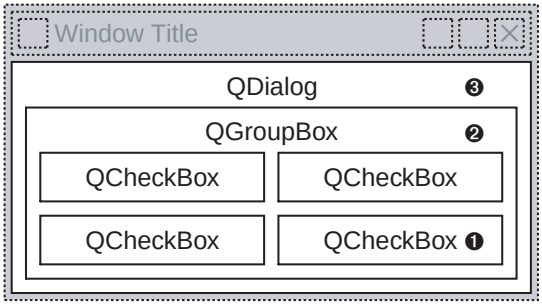


Figure 7.2
Propagation d'un événement dans une boîte de dialogue

Rester réactif pendant un traitement intensif

Quand nous appelons `QApplication::exec()`, nous démarrons une boucle d'événement de Qt. Qt émet quelques événements au démarrage pour afficher et dessiner les widgets. Puis, la boucle d'événement est exécutée, contrôlant en permanence si des événements se sont déclenchés et envoyant ces événements aux `QObject` dans l'application.

Pendant qu'un événement est traité, des événements supplémentaires peuvent être déclenchés et ajoutés à la file d'attente d'événements de Qt. Si nous passons trop de temps à traiter un événement particulier, l'interface utilisateur ne répondra plus. Par exemple, tout événement déclenché par le système de fenêtrage pendant que l'application enregistre un fichier sur le disque ne sera pas traité tant que le fichier n'a pas été sauvegardé. Pendant l'enregistrement,

l'application ne répondra pas aux requêtes du système de fenêtrage demandant le rafraîchissement de l'affichage.

Une solution consiste à utiliser plusieurs threads : un thread pour l'interface utilisateur de l'application et un autre pour accomplir la sauvegarde du fichier (ou toute autre opération de longue durée). De cette façon, l'interface utilisateur de l'application continuera à répondre pendant l'enregistrement du fichier. Nous verrons comment y parvenir dans le Chapitre 18.

Une solution plus simple consiste à appeler fréquemment `QApplication::processEvents()` dans le code de sauvegarde du fichier. Cette fonction demande à Qt de traiter tout événement en attente, puis retourne le contrôle à l'appelant. En fait, `QApplication::exec()` ne se limite pas à une simple boucle `while` autour d'un appel de fonction `processEvents()`.

Voici par exemple comment vous pourriez obtenir de l'interface utilisateur qu'elle reste réactive à l'aide de `processEvents()`, face au code de sauvegarde de fichier de l'application Spreadsheet (voir Chapitre 4) :

```
bool Spreadsheet::writeFile(const QString &fileName)
{
   QFile file(fileName);
   ...
   for (int row = 0; row < RowCount; ++row) {
      for (int column = 0; column < ColumnCount; ++column) {
         QString str = formula(row, column);
         if (!str.isEmpty())
            out << quint16(row) << quint16(column) << str;
      }
      qApp->processEvents();
   }
   return true;
}
```

Cette approche présente un risque : l'utilisateur peut fermer la fenêtre principale alors que l'application est toujours en train d'effectuer la sauvegarde, ou même cliquer sur File > Save une seconde fois, ce qui provoque un comportement indéterminé. La solution la plus simple à ce problème est de remplacer

```
qApp->processEvents();
```

par

```
qApp->processEvents(QEventLoop::ExcludeUserInputEvents);
```

qui demande à Qt d'ignorer les événements `mouse` et `key`.

Nous avons souvent besoin d'afficher un `QProgressDialog` alors qu'une longue opération se produit. `QProgressDialog` propose une barre de progression qui informe l'utilisateur de l'avancement de l'opération. `QProgressDialog` propose aussi un bouton Cancel qui permet à

l'utilisateur d'annuler l'opération. Voici le code permettant d'enregistrer une feuille de calcul avec cette approche :

```
bool Spreadsheet::writeFile(const QString &fileName)
{
   QFile file(fileName);
   ...
   QProgressDialog progress(this);
   progress.setLabelText(tr("Saving %1").arg(fileName));
   progress.setRange(0, RowCount);
   progress.setModal(true);

   for (int row = 0; row < RowCount; ++row) {
      progress.setValue(row);
      qApp->processEvents();
      if (progress.wasCanceled()) {
         file.remove();
         return false;
      }

      for (int column = 0; column < ColumnCount; ++column) {
         QString str = formula(row, column);
         if (!str.isEmpty())
            out << quint16(row) << quint16(column) << str;
      }
   }
   return true;
}
```

Nous créons un `QProgressDialog` avec `NumRows` comme nombre total d'étapes. Puis, pour chaque ligne, nous appelons `setValue()` pour mettre à jour la barre de progression. `QProgressDialog` calcule automatiquement un pourcentage en divisant la valeur actuelle d'avancement par le nombre total d'étapes. Nous invoquons `QApplication::processEvents()` pour traiter tout événement de rafraîchissement d'affichage, tout clic ou toute touche enfoncée par l'utilisateur (par exemple pour permettre à l'utilisateur de cliquer sur Cancel). Si l'utilisateur clique sur Cancel, nous annulons la sauvegarde et nous supprimons le fichier.

Nous n'appelons pas `show()` sur `QProgressDialog` parce que les boîtes de dialogue de progression le font. Si l'opération se révèle plus courte, peut-être parce que le fichier à enregistrer est petit ou parce que l'ordinateur est rapide, `QProgressDialog` le détectera et ne s'affichera pas du tout.

En complément du multithread et de l'utilisation de `QProgressDialog`, il existe une manière totalement différente de traiter les longues opérations : au lieu d'accomplir le traitement à la demande de l'utilisateur, nous pouvons ajourner ce traitement jusqu'à ce que l'application soit inactive. Cette solution peut être envisagée si le traitement peut être interrompu et repris en toute sécurité, parce que nous ne pouvons pas prédire combien de temps l'application sera inactive.

Dans Qt, cette approche peut être implémentée en utilisant un timer de 0 milliseconde. Ces timers chronomètrent dès qu'il n'y a pas d'événements en attente. Voici un exemple d'implémentation de `timerEvent()` qui présente cette approche :

```
void Spreadsheet::timerEvent(QTimerEvent *event)
{
   if (event->timerId() == myTimerId) {
      while (step < MaxStep && !qApp->hasPendingEvents()) {
         performStep(step);
         ++step;
      }
   } else {
      QTableWidget::timerEvent(event);
   }
}
```

Si `hasPendingEvents()` retourne `true`, nous interrompons le traitement et nous redonnons le contrôle à Qt. Le traitement reprendra quand Qt aura géré tous ses événements en attente.

8

Graphiques 2D et 3D

Au sommaire de ce chapitre

- ✔ Dessiner avec QPainter
- ✔ Transformations du painter
- ✔ Affichage de haute qualité avec QImage
- ✔ Impression
- ✔ Graphiques avec OpenGL

Les graphiques 2D de Qt se basent sur la classe `QPainter`. `QPainter` peut tracer des formes géométriques (points, lignes, rectangles, ellipses, arcs, cordes, segments, polygones et courbes de Bézier), de même que des objets pixmaps, des images et du texte. De plus, `QPainter` prend en charge des fonctionnalités avancées, telles que l'anticrénelage (pour les bords du texte et des formes), le mélange alpha, le remplissage dégradé et les tracés de vecteur. `QPainter` supporte aussi les transformations qui permettent de dessiner des graphiques 2D indépendants de la résolution.

`QPainter` peut également être employée pour dessiner sur un "périphérique de dessin" tel qu'un `QWidget`, `QPixmap` ou `QImage`. C'est utile quand nous écrivons des widgets personnalisés ou des classes d'éléments personnalisées avec leurs propres aspect et apparence. Il est aussi possible d'utiliser `QPainter` en association avec `QPrinter` pour imprimer et générer des fichiers PDF. Cela signifie que nous pouvons souvent nous servir du même code pour afficher des données à l'écran et pour produire des rapports imprimés.

Il existe une alternative à `QPainter` : OpenGL. OpenGL est une bibliothèque standard permettant de dessiner des graphiques 2D et 3D. Le module *QtOpenGL* facilite l'intégration de code OpenGL dans des applications Qt.

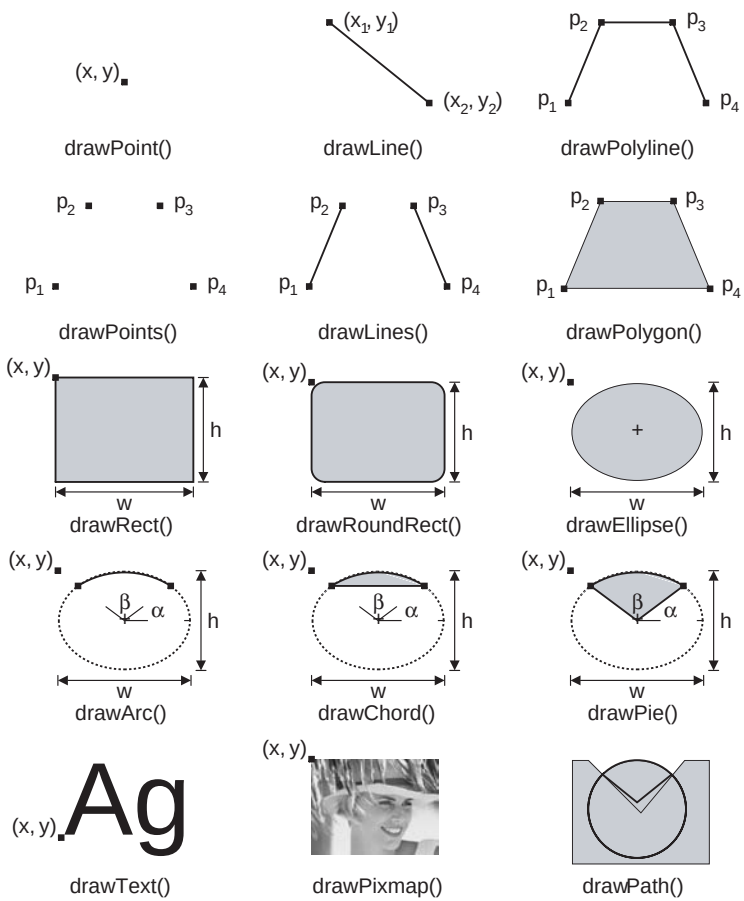
Dessiner avec QPainter

Pour commencer à dessiner sur un périphérique de dessin (généralement un widget), nous créons simplement un `QPainter` et nous transmettons un pointeur au périphérique. Par exemple :

```
void MyWidget::paintEvent(QPaintEvent *event)
{
   QPainter painter(this);
   ...
}
```

Nous avons la possibilité de dessiner différentes formes à l'aide des fonctions `draw...()` de `QPainter`. La Figure 8.1 répertorie les plus importantes. Les paramètres de `QPainter` influencent la façon de dessiner.

Figure 8.1
Les fonctions draw...() de QPainter les plus fréquemment utilisées



Certains d'entre eux proviennent du périphérique, d'autres sont initialisés à leurs valeurs par défaut. Les trois principaux paramètres sont le crayon, le pinceau et la police :

- Le *crayon* est utilisé pour tracer des lignes et les contours des formes. Il est constitué d'une couleur, d'une largeur, d'un style de trait, de capuchon et de jointure (Figures 8.2 et 8.3).
- Le *pinceau* permet de remplir des formes géométriques. Il est composé normalement d'une couleur et d'un style, mais peut également appliquer une texture (un pixmap répété à l'infini) ou un dégradé (Voir Figure 8.4).
- La *police* est utilisée pour dessiner le texte. Une police possède de nombreux attributs, dont une famille et une taille.

Figure 8.2
Styles de capuchon et de jointure

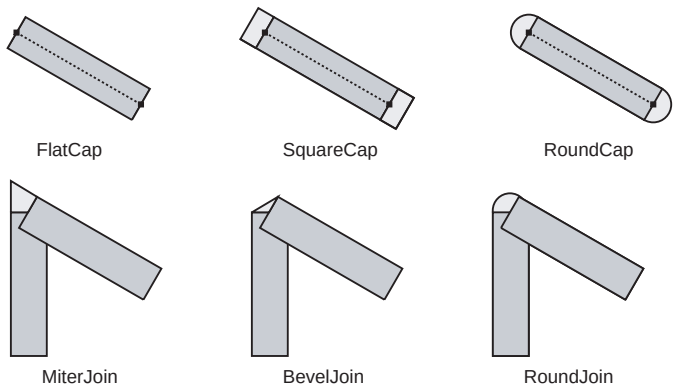


Figure 8.3
Styles de crayon

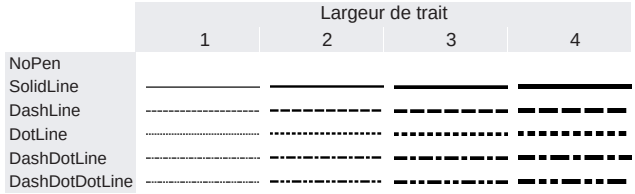
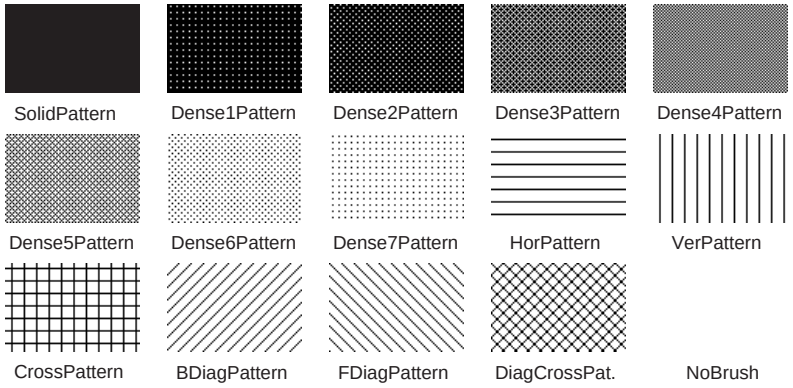


Figure 8.4
Styles prédéfinis de pinceau



Ces paramètres peuvent être modifiés à tout moment en appelant `setPen()`, `setBrush()` et `setFont()` avec un objet `QPen`, `QBrush` ou `QFont`.

Figure 8.5
Exemples de formes géométriques

Analysons quelques exemples pratiques. Voici le code permettant de dessiner l'ellipse illustrée en Figure 8.5 (a) :

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setPen(QPen(Qt::black, 12, Qt::DashDotLine, Qt::RoundCap));
painter.setBrush(QBrush(Qt::green, Qt::SolidPattern));
painter.drawEllipse(80, 80, 400, 240);
```

L'appel de `setRenderHint()` active l'anticrénelage, demandant à `QPainter` d'utiliser diverses intensités de couleur sur les bords pour réduire la distorsion visuelle qui se produit habituellement quand les contours d'une forme sont convertis en pixels. Les bords sont donc plus homogènes sur les plates-formes et les périphériques qui prennent en charge cette fonctionnalité.

Voici le code permettant de dessiner le segment illustré en Figure 8.5 (b) :

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setPen(QPen(Qt::black, 15, Qt::SolidLine, Qt::RoundCap,
                    Qt::MiterJoin));
painter.setBrush(QBrush(Qt::blue, Qt::DiagCrossPattern));
painter.drawPie(80, 80, 400, 240, 60 * 16, 270 * 16);
```

Les deux derniers arguments de `drawPie()` sont exprimés en seizièmes de degré.

Voici le code permettant de tracer la courbe de Bézier illustrée en Figure 8.5 (c) :

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);

QPainterPath path;
path.moveTo(80, 320);
path.cubicTo(200, 80, 320, 80, 480, 320);

painter.setPen(QPen(Qt::black, 8));
painter.drawPath(path);
```

La classe `QPainterPath` peut spécifier des formes vectorielles arbitraires en regroupant des éléments graphiques de base : droites, ellipses, polygones, arcs, courbes de Bézier cubiques et

quadratiques et autres tracés de dessin. Les tracés de dessin constituent la primitive graphique ultime, dans le sens où on peut désigner toute forme ou toute combinaison de formes par le terme de tracé.

Un tracé spécifie un contour, et la zone décrite par le contour peut être remplie à l'aide d'un pinceau. Dans l'exemple de la Figure 8.5 (c), nous n'avons pas utilisé de pinceau, seul le contour est donc dessiné.

Les trois exemples précédents utilisent des modèles de pinceau intégrés (`Qt::SolidPattern`, `Qt::DiagCrossPattern` et `Qt::NoBrush`). Dans les applications modernes, les remplissages dégradés représentent une alternative populaire aux remplissages monochromes. Les dégradés reposent sur une interpolation de couleur permettant d'obtenir des transitions homogènes entre deux ou plusieurs couleurs. Ils sont fréquemment utilisés pour produire des effets 3D ; par exemple, le style Plastique se sert des dégradés pour afficher des `QPushButton`.

Qt prend en charge trois types de dégradés : linéaire, conique et circulaire. L'exemple Oven Timer dans la section suivante combine les trois types de dégradés dans un seul widget pour le rendre plus réel.

- Les *dégradés linéaires* sont définis par deux points de contrôle et par une série "d'arrêts couleur" sur la ligne qui relie ces deux points. Par exemple, le dégradé linéaire de la Figure 8.6 est créé avec le code suivant :

```
QLinearGradient gradient(50, 100, 300, 350);
gradient.setColorAt(0.0, Qt::white);
gradient.setColorAt(0.2, Qt::green);
gradient.setColorAt(1.0, Qt::black);
```

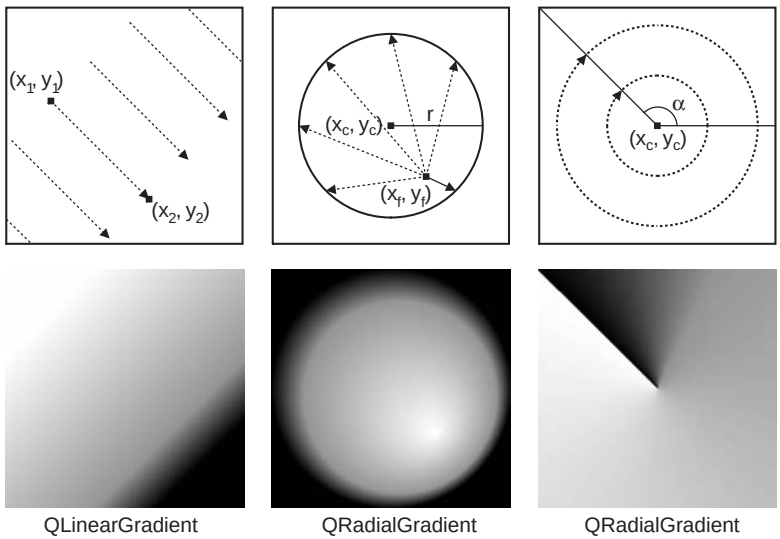


Figure 8.6
Les pinceaux dégradés de QPainter

Nous spécifions trois couleurs à trois positions différentes entre les deux points de contrôle.

Les positions sont indiquées comme des valeurs à virgule flottante entre 0 et 1, où 0 correspond au premier point de contrôle et 1 au second. Les couleurs situées entre les interruptions spécifiées sont interpolées.

- Les *dégradés circulaires* sont définis par un centre (x_c, y_c), un rayon r et une focale (x_f, y_f), en complément des interruptions de dégradé. Le centre et le rayon spécifient un cercle. Les couleurs se diffusent vers l'extérieur à partir de la focale, qui peut être le centre ou tout autre point dans le cercle.
- Les *dégradés coniques* sont définis par un centre (x_c, y_c) et un angle α. Les couleurs se diffusent autour du point central comme la trajectoire de la petite aiguille d'une montre.

Jusqu'à présent, nous avons mentionné les paramètres de crayon, de pinceau et de police de `QPainter`. En plus de ceux-ci, `QPainter` propose d'autres paramètres qui influencent la façon dont les formes et le texte sont dessinés :

- Le *pinceau de fond* est utilisé pour remplir le fond des formes géométriques (sous le modèle de pinceau), du texte ou des bitmaps quand le *mode arrière-plan* est configuré en `Qt::OpaqueMode` (la valeur par défaut est `Qt::TransparentMode`).
- *L'origine du pinceau* correspond au point de départ des modèles de pinceau, normalement le coin supérieur gauche du widget.
- La *zone d'action* est la zone du périphérique de dessin qui peut être peinte. Dessiner en dehors de cette zone n'a aucun effet.
- Le *viewport*, la *fenêtre* et la *matrice "world"* déterminent la manière dont les coordonnées logiques de `QPainter` correspondent aux coordonnées physiques du périphérique de dessin. Par défaut, celles-ci sont définies de sorte que les systèmes de coordonnées logiques et physiques coïncident. Les systèmes de coordonnées sont abordés dans la prochaine section.
- Le *mode de composition* spécifie comment les pixels qui viennent d'être dessinés doivent interagir avec les pixels déjà présents sur le périphérique de dessin. La valeur par défaut est "source over," où les pixels sont dessinés au-dessus des pixels existants. Ceci n'est pris en charge que sur certains périphériques et est traité ultérieurement dans ce chapitre.

Vous pouvez sauvegarder l'état courant d'un module de rendu nommé painter à tout moment sur une pile interne en appelant `save()` et en le restaurant plus tard en invoquant `restore()`. Cela permet par exemple de changer temporairement certains paramètres, puis de les réinitialiser à leurs valeurs antérieures, comme nous le verrons dans la prochaine section.

Transformations du painter

Avec le système de coordonnées par défaut du `QPainter`, le point (0, 0) se situe dans le coin supérieur gauche du périphérique de dessin ; les coordonnées x augmentent vers la droite et les coordonnées y sont orientées vers le bas. Chaque pixel occupe une zone d'une taille de 1×1 dans le système de coordonnées par défaut.

Il est important de comprendre que le centre d'un pixel se trouve aux coordonnées d'un "demi pixel".Par exemple, le pixel en haut à gauche couvre la zone entre les points (0, 0) et (1, 1) et son centre se trouve à (0,5, 0,5). Si nous demandons à `QPainter` de dessiner un pixel à (100, 100) par exemple, il se rapprochera du résultat en décalant les coordonnées de +0,5 dans les deux sens, le pixel sera ainsi centré sur le point (100,5, 100,5).

Cette distinction peut sembler plutôt académique de prime abord, mais elle présente des conséquences importantes en pratique. Premièrement, le décalage de +0,5 ne se produit que si l'anticrénelage est désactivé (par défaut) ; si l'anticrénelage est activé et si nous essayons de dessiner un pixel à (100, 100) en noir, `QPainter` coloriera les quatre pixels (99,5, 99,5), (99,5, 100,5), (100,5, 99,5) et (100,5, 100,5) en gris clair pour donner l'impression qu'un pixel se trouve exactement au point de rencontre de ces quatre pixels. Si cet effet ne vous plaît pas, vous pouvez l'éviter en spécifiant les coordonnées d'un demi pixel, par exemple, (100,5, 100,5).

Lorsque vous tracez des formes comme des lignes, des rectangles et des ellipses, des règles similaires s'appliquent. La Figure 8.7 vous montre comment le résultat d'un appel de `drawRect(2, 2, 6, 5)` varie en fonction de la largeur du crayon quand l'anticrénelage est désactivé. Il est notamment important de remarquer qu'un rectangle de 6×5 dessiné avec une largeur de crayon de 1 couvre en fait une zone de 7×6. C'est différent des anciens kits d'outils, y compris des versions antérieures de Qt, mais c'est essentiel pour pouvoir dessiner des images vectorielles réellement ajustables et indépendantes de la résolution.

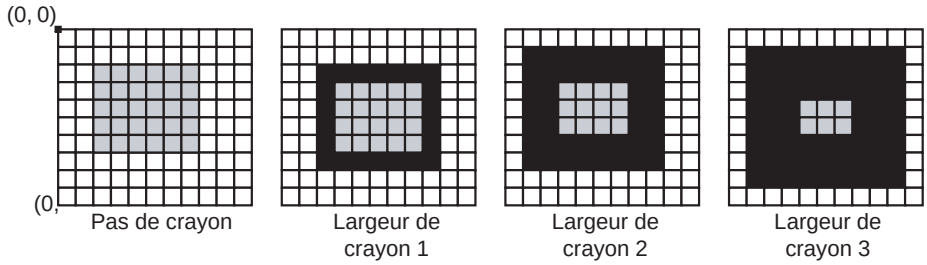


Figure 8.7
Dessiner un rectangle de 6 × 5 sans anticrénelage

Maintenant que nous avons compris le système de coordonnées par défaut, nous pouvons nous concentrer davantage sur la manière de le modifier en utilisant le viewport, la fenêtre et la matrice world de `QPainter`. (Dans ce contexte, le terme de "fenêtre" ne se réfère pas à une fenêtre au sens de widget de niveau supérieur, et le "viewport" n'a rien à voir avec le viewport de `QScrollArea`.)

Le viewport et la fenêtre sont étroitement liés. Le viewport est un rectangle arbitraire spécifié en coordonnées physiques. La fenêtre spécifie le même rectangle, mais en coordonnées logiques.

Au moment du tracé, nous indiquons des points en coordonnées logiques qui sont converties en coordonnées physiques de manière algébrique linéaire, en fonction des paramètres actuels de la fenêtre et du viewport.

Par défaut, le viewport et la fenêtre correspondent au rectangle du périphérique. Par exemple, si ce dernier est un widget de 320 × 200, le viewport et la fenêtre représentent le même rectangle de 320 × 200 avec son coin supérieur gauche à la position (0, 0). Dans ce cas, les systèmes de coordonnées logiques et physiques sont identiques.

Le mécanisme fenêtre-viewport est utile pour que le code de dessin soit indépendant de la taille ou de la résolution du périphérique de dessin. Par exemple, si nous voulons que les coordonnées logiques s'étendent de (–50, –50) à (+50, +50) avec (0, 0) au milieu, nous pouvons configurer la fenêtre comme suit :

```
painter.setWindow(-50, -50, 100, 100);
```

La paire (–50, –50) spécifie l'origine et la paire (100, 100) indique la largeur et la hauteur. Cela signifie que les coordonnées logiques (–50, –50) correspondent désormais aux coordonnées physiques (0, 0), et que les coordonnées logiques (+50, +50) correspondent aux coordonnées physiques (320, 200) (voir Figure 8.8). Dans cet exemple, nous n'avons pas modifié le viewport.

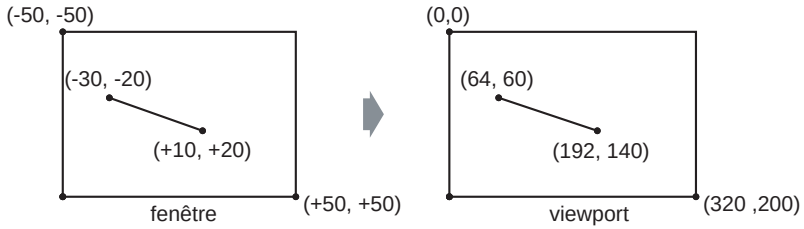


Figure 8.8
Convertir des coordonnées logiques en coordonnées physiques

Venons-en à présent à la matrice world. La matrice world est une matrice de transformation qui s'applique en plus de la conversion fenêtre-viewport. Elle nous permet de translater, mettre à l'échelle, pivoter et faire glisser les éléments que nous dessinons. Par exemple, si nous voulions dessiner un texte à un angle de 45°, nous utiliserions ce code :

```
QMatrix matrix;
matrix.rotate(45.0);
painter.setMatrix(matrix);
painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
```

Les coordonnées logiques transmises à `drawText()` sont transformées par la matrice world, puis mappées aux coordonnées physiques grâce aux paramètres fenêtre-viewport.

Si nous spécifions plusieurs transformations, elles sont appliquées dans l'ordre dans lequel nous les avons indiquées. Par exemple, si nous voulons utiliser le point (10, 20) comme pivot pour la rotation, nous pouvons translater la fenêtre, accomplir la rotation, puis translater à nouveau la fenêtre vers sa position d'origine :

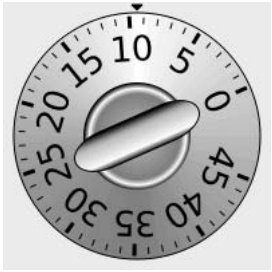
```
QMatrix matrix;
matrix.translate(-10.0, -20.0);
matrix.rotate(45.0);
matrix.translate(+10.0, +20.0);
painter.setMatrix(matrix);
painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
```

Il existe un moyen plus simple de spécifier des transformations : exploiter les fonctions pratiques `translate()`, `scale()`, `rotate()` et `shear()` de `QPainter` :

```
painter.translate(-10.0, -20.0);
painter.rotate(45.0);
painter.translate(+10.0, +20.0);
painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
```

Cependant, si nous voulons appliquer les mêmes transformations de façon répétitive, il est plus efficace de les stocker dans un objet `QMatrix` et de configurer la matrice world sur le painter dès que les transformations sont nécessaires.

Figure 8.9
Le widget OvenTimer



Pour illustrer les transformations du painter, nous allons analyser le code du widget `OvenTimer` présenté en Figure 8.9. Ce widget est conçu d'après les minuteurs de cuisine que nous utilisions avant que les fours soient équipés d'horloges intégrées. L'utilisateur peut cliquer sur un cran pour définir la durée. La molette tournera automatiquement dans le sens inverse des aiguilles d'une montre jusqu'à 0, c'est à ce moment-là que `OvenTimer` émettra le signal `timeout()`.

```
class OvenTimer : public QWidget
{
   Q_OBJECT

public:
   OvenTimer(QWidget *parent = 0);

   void setDuration(int secs);
```

```
    int duration() const;
    void draw(QPainter *painter);

signals:
    void timeout();

protected:
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);

private:
    QDateTime finishTime;
    QTimer *updateTimer;
    QTimer *finishTimer;
};
```

La classe `OvenTimer` hérite de `QWidget` et réimplémente deux fonctions virtuelles : `paintEvent()` et `mousePressEvent()`.

```
const double DegreesPerMinute = 7.0;
const double DegreesPerSecond = DegreesPerMinute / 60;
const int MaxMinutes = 45;
const int MaxSeconds = MaxMinutes * 60;
const int UpdateInterval = 1;
```

Nous définissons d'abord quelques constantes qui contrôlent l'aspect et l'apparence du minuteur de four.

```
OvenTimer::OvenTimer(QWidget *parent)
    : QWidget(parent)
{
    finishTime = QDateTime::currentDateTime();

    updateTimer = new QTimer(this);
    connect(updateTimer, SIGNAL(timeout()), this, SLOT(update()));

    finishTimer = new QTimer(this);
    finishTimer->setSingleShot(true);
    connect(finishTimer, SIGNAL(timeout()), this, SIGNAL(timeout()));
    connect(finishTimer, SIGNAL(timeout()), updateTimer, SLOT(stop()));
}
```

Dans le constructeur, nous créons deux objets QTimer : updateTimer est employé pour actualiser l'apparence du widget toutes les secondes, et finishTimer émet le signal timeout() du widget quand le minuteur du four atteint 0. finishTimer ne doit minuter qu'une seule fois, nous appelons donc setSingleShot(true) ; par défaut, les minuteurs se déclenchent de manière répétée jusqu'à ce qu'ils soient stoppés ou détruits. Le dernier appel de connect() permet d'arrêter la mise à jour du widget chaque seconde quand le minuteur est inactif.

```
void OvenTimer::setDuration(int secs)
{
    if (secs > MaxSeconds) {
```

```
        secs = MaxSeconds;
    } else if (secs <= 0) {
        secs = 0;
    }

    finishTime = QDateTime::currentDateTime().addSecs(secs);

    if (secs > 0) {
        updateTimer->start(UpdateInterval * 1000);
        finishTimer->start(secs * 1000);
    } else {
        updateTimer->stop();
        finishTimer->stop();
    }
    update();
}
```

La fonction `setDuration()` définit la durée du minuteur du four en nombre donné de secondes. Nous calculons l'heure de fin en ajoutant la durée à l'heure courante (obtenue à partir de `QDateTime::currentDateTime()`) et nous la stockons dans la variable privée `finishTime`. Nous finissons en invoquant `update()` pour redessiner le widget avec la nouvelle durée.

La variable `finishTime` est de type `QDateTime`. Vu que la variable contient une date et une heure, nous évitons ainsi tout bogue lorsque l'heure courante se situe avant minuit et l'heure de fin après minuit.

```
int OvenTimer::duration() const
{
    int secs = QDateTime::currentDateTime().secsTo(finishTime);
    if (secs < 0)
        secs = 0;
    return secs;
}
```

La fonction `duration()` retourne le nombre de secondes restantes avant que le minuteur ne s'arrête. Si le minuteur est inactif, nous retournons 0.

```
void OvenTimer::mousePressEvent(QMouseEvent *event)
{
    QPointF point = event->pos() - rect().center();
    double theta = atan2(-point.x(), -point.y()) * 180 / 3.14159265359;
    setDuration(duration() + int(theta / DegreesPerSecond));
    update();
}
```

Si l'utilisateur clique sur le widget, nous recherchons le cran le plus proche grâce à une formule mathématique subtile mais efficace, et nous utilisons le résultat pour définir la nouvelle durée. Puis nous planifions un réaffichage. Le cran sur lequel l'utilisateur a cliqué sera désormais en haut et se déplacera dans le sens inverse des aiguilles d'une montre au fur et à mesure que le temps s'écoule jusqu'à atteindre 0.

```
void OvenTimer::paintEvent(QPaintEvent * /* event */)
{
   QPainter painter(this);
   painter.setRenderHint(QPainter::Antialiasing, true);

   int side = qMin(width(), height());

   painter.setViewport((width() - side) / 2, (height() - side) / 2,
                       side, side);
   painter.setWindow(-50, -50, 100, 100);

   draw(&painter);
}
```

Dans `paintEvent()`, nous définissons le viewport de sorte qu'il devienne le carré le pus grand qui peut entrer dans le widget et nous définissons la fenêtre en rectangle (–50, –50, 100, 100), c'est-à-dire le rectangle de 100 _ 100 allant de (–50, –50) à (+50, +50). La fonction modèle `qMin()` retourne le plus bas de ses deux arguments. Nous appelons ensuite la fonction `draw()` qui se chargera du dessin.

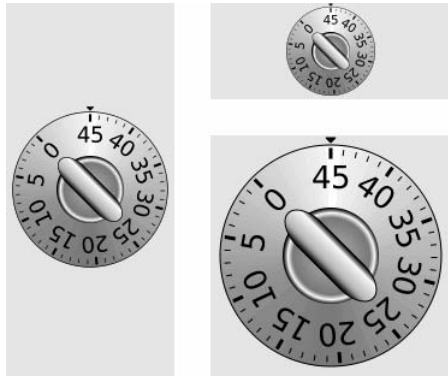


Figure 8.10
Le widget OvenTimer en trois tailles différentes

Si nous n'avions pas défini le viewport en carré, le minuteur du four se transformerait en ellipse quand le widget serait redimensionné en rectangle (non carré). Pour éviter de telles déformations, nous devons configurer le viewport et la fenêtre en rectangles ayant le même format d'image.

Analysons à présent le code de dessin :

```
void OvenTimer::draw(QPainter *painter)
{
   static const int triangle[3][2] = {
      { -2, -49 }, { +2, -49 }, { 0, -47 }
   };
   QPen thickPen(palette().foreground(), 1.5);
   QPen thinPen(palette().foreground(), 0.5);
   QColor niceBlue(150, 150, 200);
```

```
painter->setPen(thinPen);
painter->setBrush(palette().foreground());
painter->drawPolygon(QPolygon(3, &triangle[0][0]));
```

Nous commençons par dessiner le petit triangle qui symbolise la position 0 en haut du widget. Le triangle est spécifié par trois coordonnées codées et nous utilisons `drawPolygon()` pour l'afficher.

L'aspect pratique du mécanisme fenêtre-viewport, c'est que nous avons la possibilité de coder les coordonnées utilisées dans les commandes de dessin et de toujours obtenir un bon comportement lors du redimensionnement.

```
QConicalGradient coneGradient(0, 0, -90.0);
coneGradient.setColorAt(0.0, Qt::darkGray);
coneGradient.setColorAt(0.2, niceBlue);
coneGradient.setColorAt(0.5, Qt::white);
coneGradient.setColorAt(1.0, Qt::darkGray);

painter->setBrush(coneGradient);
painter->drawEllipse(-46, -46, 92, 92);
```

Nous traçons le cercle extérieur et nous le remplissons avec un dégradé conique. Le centre du dégradé se trouve à la position (0, 0) et son angle est de –90°.

```
QRadialGradient haloGradient(0, 0, 20, 0, 0);
haloGradient.setColorAt(0.0, Qt::lightGray);
haloGradient.setColorAt(0.8, Qt::darkGray);
haloGradient.setColorAt(0.9, Qt::white);
haloGradient.setColorAt(1.0, Qt::black);

painter->setPen(Qt::NoPen);
painter->setBrush(haloGradient);
painter->drawEllipse(-20, -20, 40, 40);
```

Nous nous servons d'un dégradé radial pour le cercle intérieur. Le centre et la focale du dégradé se situent à (0, 0). Le rayon du dégradé est égal à 20.

```
QLinearGradient knobGradient(-7, -25, 7, -25);
knobGradient.setColorAt(0.0, Qt::black);
knobGradient.setColorAt(0.2, niceBlue);
knobGradient.setColorAt(0.3, Qt::lightGray);
knobGradient.setColorAt(0.8, Qt::white);
knobGradient.setColorAt(1.0, Qt::black);

painter->rotate(duration() * DegreesPerSecond);
painter->setBrush(knobGradient);
painter->setPen(thinPen);
painter->drawRoundRect(-7, -25, 14, 50, 150, 50);

for (int i = 0; i <= MaxMinutes; ++i) {
   if (i % 5 == 0) {
```

```
            painter->setPen(thickPen);
            painter->drawLine(0, -41, 0, -44);
            painter->drawText(-15, -41, 30, 25,
                              Qt::AlignHCenter | Qt::AlignTop,
                              QString::number(i));
        } else {
            painter->setPen(thinPen);
            painter->drawLine(0, -42, 0, -44);
        }
        painter->rotate(-DegreesPerMinute);
    }
}
```

Nous appelons `rotate()` pour faire pivoter le système de coordonnées du painter. Dans l'ancien système de coordonnées, la marque correspondant à 0 minute se trouvait en haut ; maintenant, elle se déplace vers l'endroit approprié pour le temps restant. Nous dessinons le bouton rectangulaire après la rotation, parce que son orientation dépend de l'angle de cette rotation.

Dans la boucle `for`, nous dessinons les graduations tout autour du cercle extérieur et les nombres pour chaque multiple de 5 minutes. Le texte est dessiné dans un rectangle invisible en dessous de la graduation. A la fin de chaque itération, nous faisons pivoter le painter dans le sens des aiguilles d'une montre de 7°, ce qui correspond à une minute. La prochaine fois que nous dessinons une graduation, elle sera à une position différente autour du cercle, même si les coordonnées transmises aux appels de `drawLine()` et `drawText()` sont toujours les mêmes.

Le code de la boucle `for` souffre d'un défaut mineur qui deviendrait rapidement apparent si nous effectuions davantage d'itérations. A chaque fois que nous appelons `rotate()`, nous multiplions la matrice courante par une matrice de rotation, engendrant ainsi une nouvelle matrice world. Les problèmes d'arrondis associés à l'arithmétique en virgule flottante entraînent une matrice world très imprécise. Voici un moyen de réécrire le code pour éviter ce problème, en exécutant `save()` et `restore()` pour sauvegarder et recharger la matrice de transformation originale pour chaque itération :

```
    for (int i = 0; i <= MaxMinutes; ++i) {
        painter->save();
        painter->rotate(-i * DegreesPerMinute);

        if (i % 5 == 0) {
            painter->setPen(thickPen);
            painter->drawLine(0, -41, 0, -44);
            painter->drawText(-15, -41, 30, 25,
                              Qt::AlignHCenter | Qt::AlignTop,
                              QString::number(i));
        } else {
            painter->setPen(thinPen);
            painter->drawLine(0, -42, 0, -44);
        }
        painter->restore();
    }
```

Il existe un autre moyen d'implémenter un minuteur de four : calculer les positions (*x*, *y*) soi-même, en utilisant `sin()` et `cos()` pour trouver les positions autour du cercle. Mais nous aurions encore dû employer une translation et une rotation pour dessiner le texte à un certain angle.

Affichage de haute qualité avec QImage

Au moment de dessiner, vous pourriez avoir à trouver un compromis entre vitesse et précision. Par exemple, sous X11 et Mac OS X, le dessin sur un `QWidget` ou un `QPixmap` repose sur le moteur de dessin natif de la plate-forme. Sous X11, ceci réduit au maximum les communications avec le serveur X ; seules les commandes de dessin sont envoyées plutôt que les données de l'image. Le principal inconvénient de cette approche, c'est que le champ d'action de Qt est limité à celui de la prise en charge native de la plate-forme :

- Sous X11, des fonctionnalités, telles que l'anticrénelage et le support des coordonnées fractionnaires, ne sont disponibles que si l'extension X Render se trouve sur le serveur X.
- Sous Mac OS X, le moteur graphique natif crénelé s'appuie sur des algorithmes différents pour dessiner des polygones par rapport à X11 et Windows, avec des résultats légèrement différents.

Quand la précision est plus importante que l'efficacité, nous pouvons dessiner un `QImage` et copier le résultat à l'écran. Celui-ci utilise toujours le moteur de dessin interne de Qt, aboutissant ainsi à des résultats identiques sur toutes les plates-formes. La seule restriction, c'est que le `QImage`, sur lequel nous dessinons, doit être créé avec un argument de type `QImage::Format_RGB32` ou `QImage::Format_ARGB32_Premultiplied`.

Le format ARGB32 prémultiplié est presque identique au format traditionnel ARGB32 (`0xaarrggbb`), à la différence près que les canaux rouge, vert et bleu sont "prémultipliés" par le canal alpha. Cela signifie que les valeurs RVB, qui s'étendent normalement de `0x00` à `0xFF`, sont mises à l'échelle de `0x00` à la valeur alpha. Par exemple, une couleur bleue transparente à 50 % est représentée par `0x7F0000FF` en format ARGB32, mais par `0x7F00007F` en format ARGB32 prémultiplié. De même, une couleur vert foncé transparente à 75 % représentée par `0x3F008000` en format ARGB32 deviendrait `0x3F002000` en format ARGB32 prémultiplié.

Supposons que nous souhaitons utiliser l'anticrénelage pour dessiner un widget et que nous voulons obtenir de bons résultats même sous des systèmes X11 sans l'extension X Render. Voici la syntaxe du gestionnaire `paintEvent()` d'origine, qui se base sur X Render pour l'anticrénelage :

```
void MyWidget::paintEvent(QPaintEvent *event)
{
   QPainter painter(this);
   painter.setRenderHint(QPainter::Antialiasing, true);
   draw(&painter);
}
```

Voilà comment réécrire la fonction `paintEvent()` du widget pour exploiter le moteur graphique de Qt indépendant de la plate-forme :

```
void MyWidget::paintEvent(QPaintEvent *event)
{
   QImage image(size(), QImage::Format_ARGB32_Premultiplied);
   QPainter imagePainter(&image);
   imagePainter.initFrom(this);
   imagePainter.setRenderHint(QPainter::Antialiasing, true);
   imagePainter.eraseRect(rect());
   draw(&imagePainter);
   imagePainter.end();

   QPainter widgetPainter(this);
   widgetPainter.drawImage(0, 0, image);
}
```

Nous créons un `QImage` de la même taille que le widget en format ARGB32 prémultiplié, et un `QPainter` pour dessiner sur l'image. L'appel de `initFrom()` initialise le crayon, le fond et la police en fonction du widget. Nous effectuons notre dessin avec `QPainter` comme d'habitude, et à la fin, nous réutilisons l'objet `QPainter` pour copier l'image sur le widget.

Cette approche produit d'excellents résultats identiques sur toutes les plates-formes, à l'exception de l'affichage de la police qui dépend des polices installées.

Une fonctionnalité particulièrement puissante du moteur graphique de Qt est sa prise en charge des modes de composition. Ceux-ci spécifient comment un pixel source et un pixel de destination fusionnent pendant le dessin. Ceci s'applique à toutes les opérations de dessin, y compris le crayon, le pinceau, le dégradé et l'image.

Le mode de composition par défaut est `QImage::CompositionMode_SourceOver`, ce qui signifie que le pixel source (celui que nous dessinons) remplace le pixel de destination (le pixel existant) de telle manière que le composant alpha de la source définit sa translucidité. Dans la Figure 8.11, vous voyez un papillon à moitié transparent dessiné sur un motif à damiers avec les différents modes.

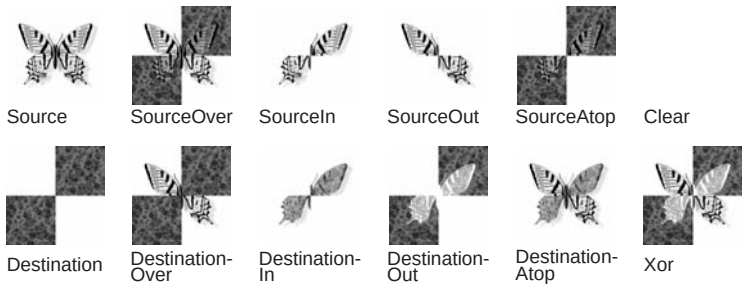


Figure 8.11
Les modes de composition de QPainter

Les modes de composition sont définis à l'aide de `QPainter::setCompositionMode()`. Par exemple, voici comment créer un `QImage` qui combine en XOR le papillon et le motif à damiers :

```
QImage resultImage = checkerPatternImage;
QPainter painter(&resultImage);
painter.setCompositionMode(QPainter::CompositionMode_Xor);
painter.drawImage(0, 0, butterflyImage);
```

Il faut être conscient du problème lié au fait que l'opération `QImage::CompositionMode_Xor` s'applique au canal alpha. Cela signifie que si nous appliquons XOR (OU exclusif) à la couleur blanche (`0xFFFFFFFF`), nous obtenons une couleur transparente (`0x00000000`), et pas noire (`0xFF000000`).

Impression

L'impression dans Qt est équivalente au dessin sur `QWidget`, `QPixmap` ou `QImage`. Plusieurs étapes sont nécessaires :

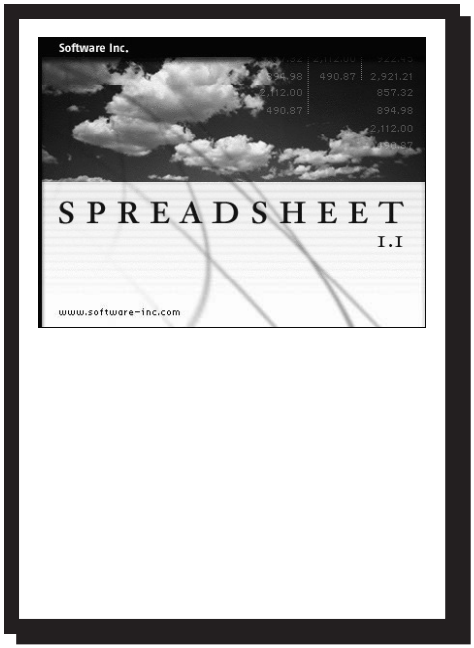
1. créer un `QPrinter` qui fera office de périphérique de dessin ;
2. ouvrir un `QPrintDialog`, qui permet à l'utilisateur de choisir une imprimante et de configurer certaines options ;
3. créer un `QPainter` pour agir sur le `QPrinter` ;
4. dessiner une page à l'aide de `QPainter` ;
5. appeler `QPrinter::newPage()` pour passer à la page suivante ;
6. répéter les étapes 4 et 5 jusqu'à ce que toutes les pages soient imprimées.

Sous Windows et Mac OS X, `QPrinter` utilise les pilotes d'imprimante du système. Sous Unix, il génère un PostScript et l'envoie à `lp` ou `lpr` (ou au programme défini en exécutant `QPrinter::setPrintProgram()`). `QPrinter` peut aussi servir à générer des fichiers PDF en appelant `setOutputFormat(QPrinter::PdfFormat)`.

Commençons par quelques exemples simples qui s'impriment tous sur une seule page. Le premier exemple imprime un `QImage`(voir Figure 8.12) :

```
void PrintWindow::printImage(const QImage &image)
{
   QPrintDialog printDialog(&printer, this);
   if (printDialog.exec()) {
      QPainter painter(&printer);
      QRect rect = painter.viewport();
      QSize size = image.size();
      size.scale(rect.size(), Qt::KeepAspectRatio);
      painter.setViewport(rect.x(), rect.y(),
                          size.width(), size.height());
      painter.setWindow(image.rect());
      painter.drawImage(0, 0, image);
   }
}
```

Figure 8.12
Imprimer un QImage



Nous supposons que la classe `PrintWindow` possède une variable membre appelée `printer` de type `QPrinter`. Nous aurions simplement pu créer `QPrinter` sur la pile dans `printImage()`, mais les paramètres de l'utilisateur auraient été perdus entre deux impressions.

Nous créons un `QPrintDialog` et nous invoquons `exec()` pour l'afficher. Il retourne `true` si l'utilisateur a cliqué sur le bouton OK ; sinon il retourne `false`. Après l'appel de `exec()`, l'objet `QPrinter` est prêt à être utilisé. (Il est aussi possible d'imprimer sans utiliser `QPrintDialog`, en appelant directement des fonctions membres de `QPrinter` pour configurer les divers aspects.)

Nous créons ensuite un `QPainter` pour dessiner sur le `QPrinter`. Nous définissons la fenêtre en rectangle de l'image et le viewport en un rectangle du même format d'image, puis nous dessinons l'image à la position (0, 0).

Par défaut, la fenêtre de `QPainter` est initialisée de sorte que l'imprimante semble avoir une résolution similaire à l'écran (en général entre 72 et 100 points par pouce), ce qui facilite la réutilisation du code de dessin du widget pour l'impression. Ici, ce n'était pas un problème, parce que nous avons défini notre propre fenêtre.

Imprimer des éléments qui ne s'étendent pas sur plus d'une page se révèle très simple, mais de nombreuses applications ont besoin d'imprimer plusieurs pages. Pour celles-ci, nous devons dessiner une page à la fois et appeler `newPage()` pour passer à la page suivante.

Ceci soulève un problème : déterminer la quantité d'informations que nous pouvons imprimer sur chaque page. Il existe deux approches principales pour gérer les documents multipages avec Qt :

- Nous pouvons convertir nos données en format HTML et les afficher avec `QTextDocument`, le moteur de texte de Qt.
- Nous pouvons effectuer le dessin et la répartition sur les pages manuellement.

Nous allons analyser les deux approches. En guise d'exemple, nous allons imprimer un guide des fleurs : une liste de noms de fleurs, chacun comprenant une description sous forme de texte. Chaque entrée du guide est stockée sous forme de chaîne au format "*nom: description*," par exemple :

```
Miltonopsis santanae: une des espèces d'orchidées les plus dangereuses.
```

Vu que les données relatives à chaque fleur sont représentées par une seule chaîne, nous pouvons représenter toutes les fleurs dans le guide avec un `QStringList`. Voici la fonction qui imprime le guide des fleurs au moyen du moteur de texte de Qt :

```
void PrintWindow::printFlowerGuide(const QStringList &entries)
{
   QString html;

   foreach (QString entry, entries) {
      QStringList fields = entry.split(": ");
      QString title = Qt::escape(fields[0]);
      QString body = Qt::escape(fields[1]);

      html += "<table width=\"100%\" border=1 cellspacing=0>\n"
              "<tr><td bgcolor=\"lightgray\"><font size=\"+1\">"
              "<b><i>" + title + "</i></b></font>\n<tr><td>" + body
              + "\n</table>\n<br>\n";
   }
   printHtml(html);
}
```

La première étape consiste à convertir `QStringList` en format HTML. Chaque fleur est représentée par un tableau HTML avec deux cellules. Nous exécutons `Qt::escape()` pour remplacer les caractères spéciaux "&", "<", ">" par les entités HTML correspondantes ("&", "<", ">"). Nous appelons ensuite `printHtml()` pour imprimer le texte.

```
void PrintWindow::printHtml(const QString &html)
{
   QPrintDialog printDialog(&printer, this);
   if (printDialog.exec()) {
       QPainter painter(&printer);
       QTextDocument textDocument;
       textDocument.setHtml(html);
       textDocument.print(&printer);
   }
}
```

La fonction `printHtml()` ouvre un `QPrintDialog` et se charge d'imprimer un document HTML. Elle peut être réutilisée "telle quelle" dans n'importe quelle application Qt pour imprimer des pages HTML arbitraires.

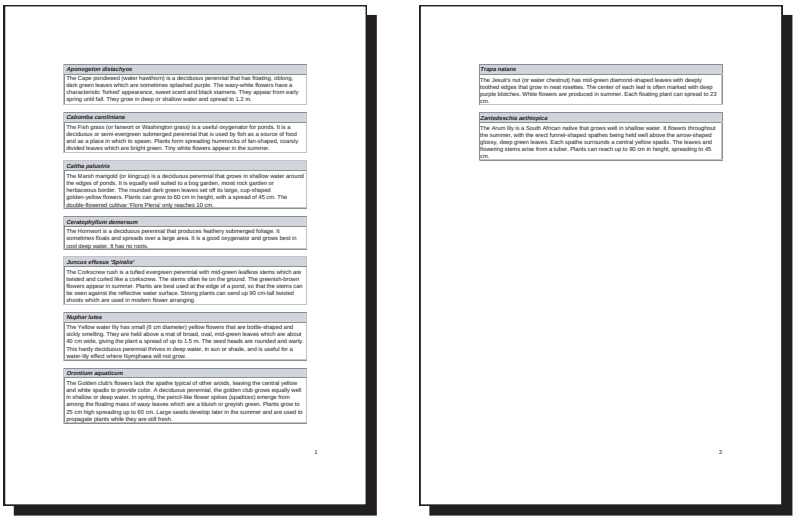


Figure 8.13
Imprimer un guide des fleurs avec QTextDocument

Convertir un document au format HTML et utiliser `QTextDocument` pour l'imprimer est de loin la méthode la plus pratique pour imprimer des rapports et d'autres documents complexes. Dès que vous avez besoin d'un niveau de contrôle supérieur, vous pouvez envisager de gérer la mise en page et le dessin manuellement. Voyons maintenant comment nous pouvons utiliser cette approche pour imprimer le guide des fleurs (voir Figure 8.13). Voilà la nouvelle fonction `printFlowerGuide()` :

```
void PrintWindow::printFlowerGuide(const QStringList &entries)
{
   QPrintDialog printDialog(&printer, this);
   if (printDialog.exec()) {
      QPainter painter(&printer);
      QList<QStringList> pages;

      paginate(&painter, &pages, entries);
      printPages(&painter, pages);
   }
}
```

Après avoir configuré l'imprimante et construit le painter, nous appelons la fonction `paginate()` pour déterminer quelle entrée doit apparaître sur quelle page. Vous obtenez donc une liste de `QStringList`, chacun contenant les entrées d'une page. Nous transmettons ce résultat à `printPages()`.

Supposons, par exemple, que le guide des fleurs contient 6 entrées, que nous appellerons A, B, C, D, E et F. Imaginons maintenant qu'il y a suffisamment de place pour A et B sur la première page, pour C, D et E sur la deuxième page et pour F sur la troisième page. La liste `pages` contiendrait donc la liste `[A, B]` à la position d'index 0, la liste `[C, D, E]` à la position d'index 1 et la liste `[F]` à la position d'index 2.

```
void PrintWindow::paginate(QPainter *painter, QList<QStringList> *pages,
                           const QStringList &entries)
{
   QStringList currentPage;
   int pageHeight = painter->window().height() - 2 * LargeGap;
   int y = 0;

   foreach (QString entry, entries) {
      int height = entryHeight(painter, entry);
      if (y + height > pageHeight && !currentPage.empty()) {
          pages->append(currentPage);
          currentPage.clear();
          y = 0;
      }
      currentPage.append(entry);
      y += height + MediumGap;
   }
   if (!currentPage.empty())
       pages->append(currentPage);
}
```

La fonction `paginate()` répartit les entrées du guide des fleurs sur les pages. Elle se base sur la fonction `entryHeight()` qui calcule la hauteur d'une entrée. Elle tient également compte des espaces vides verticaux en haut et en bas de la page, de taille `LargeGap`.

Nous parcourons les entrées et nous les ajoutons à la page en cours jusqu'à ce qu'une entrée n'ait plus suffisamment de place sur cette page ; puis nous ajoutons la page en cours à la liste `pages` et nous commençons une nouvelle page.

```
int PrintWindow::entryHeight(QPainter *painter, const QString &entry)
{
   QStringList fields = entry.split(": ");
   QString title = fields[0];
   QString body = fields[1];

   int textWidth = painter->window().width() - 2 * SmallGap;
   int maxHeight = painter->window().height();

   painter->setFont(titleFont);
   QRect titleRect = painter->boundingRect(0, 0, textWidth, maxHeight,
                                           Qt::TextWordWrap, title);
   painter->setFont(bodyFont);
   QRect bodyRect = painter->boundingRect(0, 0, textWidth, maxHeight,
                                          Qt::TextWordWrap, body);
   return titleRect.height() + bodyRect.height() + 4 * SmallGap;
}
```

La fonction `entryHeight()` se sert de `QPainter::boundingRect()` pour calculer l'espace vertical nécessaire pour une entrée. La Figure 8.14 présente la disposition d'une entrée et la signification des constantes `SmallGap` et `MediumGap`.

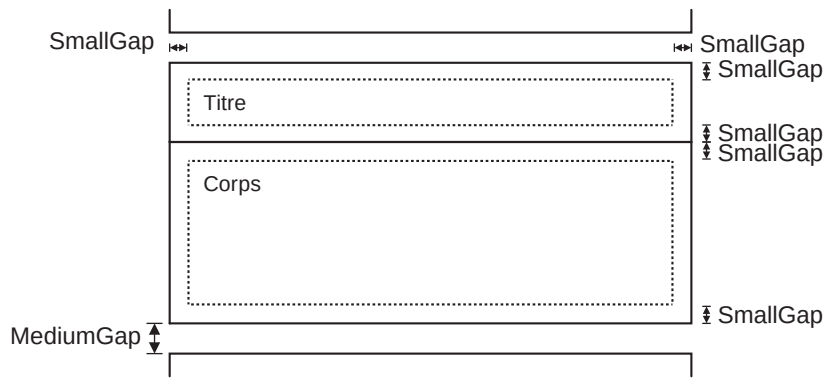


Figure 8.14
La disposition d'une entrée

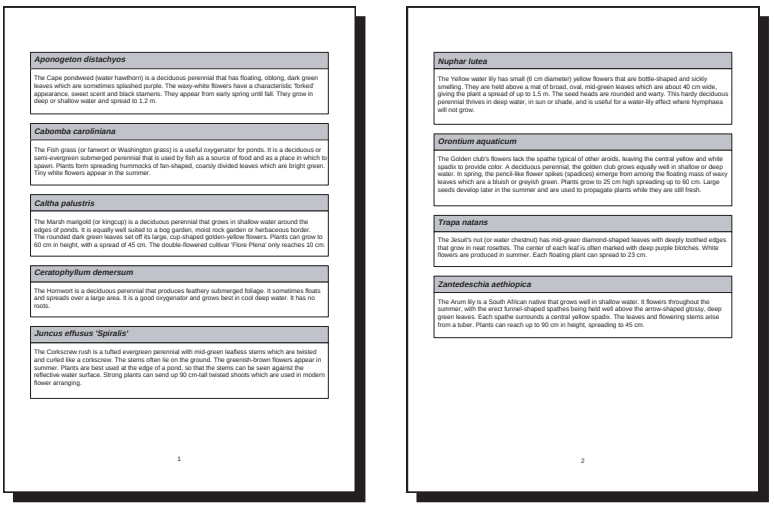
```
void PrintWindow::printPages(QPainter *painter,
                             const QList<QStringList> &pages)
{
   int firstPage = printer.fromPage() - 1;
   if (firstPage >= pages.size())
       return;
   if (firstPage == -1)
       firstPage = 0;

   int lastPage = printer.toPage() - 1;
   if (lastPage == -1 || lastPage >= pages.size())
       lastPage = pages.size() - 1;

   int numPages = lastPage - firstPage + 1;

   for (int i = 0; i < printer.numCopies(); ++i) {
       for (int j = 0; j < numPages; ++j) {
           if (i != 0 || j != 0)
           printer.newPage();

         int index;
         if (printer.pageOrder() == QPrinter::FirstPageFirst) {
             index = firstPage + j;
         } else {
             index = lastPage - j;
         }
         printPage(painter, pages[index], index + 1);
      }
   }
}
```

Le rôle de la fonction `printPages()` est d'imprimer chaque page à l'aide de `printPage()` dans le bon ordre et les bonnes quantités. Grâce à `QPrintDialog`, l'utilisateur peut demander plusieurs copies, spécifier une plage d'impression ou demander les pages en ordre inverse. C'est à nous d'honorer ces options – ou de les désactiver au moyen de `QPrintDialog::setEnabledOptions()`.

Nous déterminons tout d'abord la plage à imprimer. Les fonctions `fromPage()` et `toPage()` de `QPrinter` retournent les numéros de page sélectionnés par l'utilisateur, ou 0 si aucune plage n'a été choisie. Nous soustrayons 1, parce que notre liste `pages` est indexée à partir de 0, et nous définissons `firstPage` et `lastPage` de sorte à couvrir la totalité de la plage si l'utilisateur n'a rien précisé.

Puis nous imprimons chaque page. La boucle externe `for` effectue une itération autant de fois que nécessaire pour produire le nombre de copies demandé par l'utilisateur. La plupart des pilotes d'imprimante prennent en charge les copies multiples, `QPrinter::numCopies()` retourne toujours 1 pour celles-ci. Si le pilote ne peut pas gérer plusieurs copies, `numCopies()` renvoie le nombre de copies demandé par l'utilisateur, et l'application se charge d'imprimer ce nombre de copies. (Dans l'exemple `QImage` précédent, nous avons ignoré `numCopies()` pour une question de simplicité.)

Figure 8.15
Imprimer un guide des fleurs avec QPainter



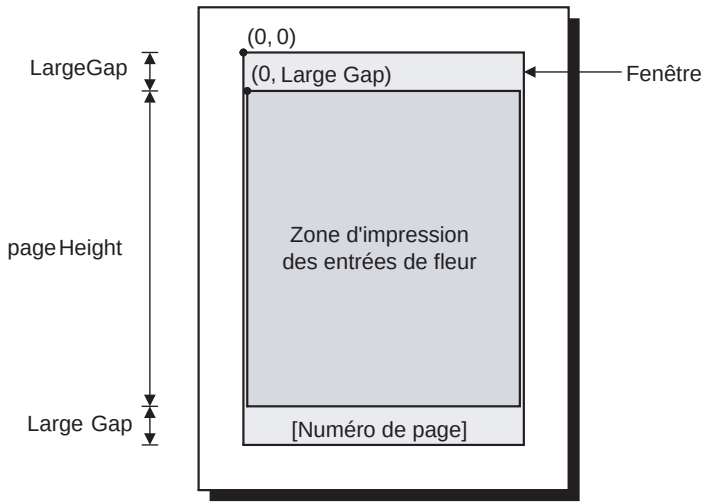
La boucle interne `for` parcourt les pages. Si la page n'est pas la première page, nous appelons `newPage()` pour supprimer l'ancienne page de la mémoire et pour commencer à dessiner sur une nouvelle page. Nous invoquons `printPage()` pour dessiner chaque page.

```
void PrintWindow::printPage(QPainter *painter,
                            const QStringList &entries, int pageNumber)
{
   painter->save();
```

```
    painter->translate(0, LargeGap);
    foreach (QString entry, entries) {
        QStringList fields = entry.split(": ");
        QString title = fields[0];
        QString body = fields[1];
        printBox(painter, title, titleFont, Qt::lightGray);
        printBox(painter, body, bodyFont, Qt::white);
        painter->translate(0, MediumGap);
    }
    painter->restore();

    painter->setFont(footerFont);
    painter->drawText(painter->window(),
                      Qt::AlignHCenter | Qt::AlignBottom,
                      QString::number(pageNumber));
}
```

La fonction `printPage()` parcourt toutes les entrées du guide des fleurs et les imprime grâce à deux appels de `printBox()` : un pour le titre (le nom de la fleur) et un pour le corps (sa description). Elle dessine également le numéro de page centré au bas de la page (voir Figure 8.16).

Figure 8.16
La disposition d'une page du guide des fleurs




```
void PrintWindow::printBox(QPainter *painter, const QString &str,
                           const QFont &font, const QBrush &brush)
{
    painter->setFont(font);

    int boxWidth = painter->window().width();
    int textWidth = boxWidth - 2 * SmallGap;
    int maxHeight = painter->window().height();
```

```
    QRect textRect = painter->boundingRect(SmallGap, SmallGap,
                                           textWidth, maxHeight,
                                           Qt::TextWordWrap, str);
    int boxHeight = textRect.height() + 2 * SmallGap;

    painter->setPen(QPen(Qt::black, 2, Qt::SolidLine));
    painter->setBrush(brush);
    painter->drawRect(0, 0, boxWidth, boxHeight);
    painter->drawText(textRect, Qt::TextWordWrap, str);
    painter->translate(0, boxHeight);
}
```

La fonction `printBox()` trace les contours d'une boîte, puis dessine le texte à l'intérieur.

Graphiques avec OpenGL

OpenGL est une API standard permettant d'afficher des graphiques 2D et 3D. Les applications Qt peuvent tracer des graphiques 3D en utilisant le module *QtOpenGL*, qui se base sur la bibliothèque OpenGL du système. Cette section suppose que vous connaissez déjà OpenGL. Si vous découvrez OpenGL, consultez d'abord **http://www.opengl.org/** pour en apprendre davantage.

Figure 8.17
L'application Tetrahedron

Dessiner des graphiques avec OpenGL dans une application Qt se révèle très facile : vous devez dériver `QGLWidget`, réimplémenter quelques fonctions virtuelles et relier l'application aux bibliothèques *QtOpenGL* et OpenGL. Etant donné que `QGLWidget` hérite de `QWidget`, la majorité des notions que nous connaissons déjà peuvent s'appliquer à ce cas. La principale différence c'est que nous utilisons des fonctions OpenGL standards pour dessiner en lieu et place de `QPainter`.

Pour vous montrer comment cela fonctionne, nous allons analyser le code de l'application Tetrahedron présentée en Figure 8.17. L'application présente un tétraèdre en 3D, ou une matrice à quatre faces, chaque face ayant une couleur différente. L'utilisateur peut faire pivoter le tétraèdre en appuyant sur un bouton de la souris et en le faisant glisser. Il peut aussi définir la couleur d'une face en double-cliquant dessus et en choisissant une couleur dans le `QColorDialog` qui s'ouvre.

```
class Tetrahedron : public QGLWidget
{
   Q_OBJECT

public:
   Tetrahedron(QWidget *parent = 0);

protected:
   void initializeGL();
   void resizeGL(int width, int height);
   void paintGL();
   void mousePressEvent(QMouseEvent *event);
   void mouseMoveEvent(QMouseEvent *event);
   void mouseDoubleClickEvent(QMouseEvent *event);

private:
   void draw();
   int faceAtPosition(const QPoint &pos);

   GLfloat rotationX;
   GLfloat rotationY;
   GLfloat rotationZ;
   QColor faceColors[4];
   QPoint lastPos;
};
```

La classe `Tetrahedron` hérite de `QGLWidget`. Les fonctions `initializeGL()`, `resizeGL()` et `paintGL()` sont réimplémentées dans `QGLWidget`. Les gestionnaires d'événements `mouse` sont réimplémentés dans `QWidget` comme d'habitude.

```
Tetrahedron::Tetrahedron(QWidget *parent)
   : QGLWidget(parent)
{
   setFormat(QGLFormat(QGL::DoubleBuffer | QGL::DepthBuffer));
   rotationX = -21.0;
   rotationY = -57.0;
   rotationZ = 0.0;
   faceColors[0] = Qt::red;
   faceColors[1] = Qt::green;
   faceColors[2] = Qt::blue;
   faceColors[3] = Qt::yellow;
}
```

Dans le constructeur, nous appelons `QGLWidget::setFormat()` pour spécifier le contexte d'affichage OpenGL et nous initialisons les variables privées de la classe.

```
void Tetrahedron::initializeGL()
{
   qglClearColor(Qt::black);
   glShadeModel(GL_FLAT);
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_CULL_FACE);
}
```

La fonction `initializeGL()` est invoquée une seule fois avant que `paintGL()` soit appelée. C'est donc dans le code de cette fonction que nous allons configurer le contexte d'affichage OpenGL, définir les listes d'affichage et accomplir d'autres initialisations.

Tout le code est en OpenGL standard, sauf l'appel de la fonction `qglClearColor()` de `QGLWidget`. Si nous voulions uniquement du code OpenGL standard, nous aurions pu appeler `glClearColor()` en mode RGBA et `glClearIndex()` en mode table des couleurs.

```
void Tetrahedron::resizeGL(int width, int height)
{
   glViewport(0, 0, width, height);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   GLfloat x = GLfloat(width) / height;
   glFrustum(-x, x, -1.0, 1.0, 4.0, 15.0);
   glMatrixMode(GL_MODELVIEW);
}
```

La fonction `resizeGL()` est invoquée avant le premier appel de `paintGL()`, mais après l'appel de `initializeGL()`. Elle est aussi invoquée dès que le widget est redimensionné. C'est là que nous avons la possibilité de configurer le viewport OpenGL, la projection et tout autre paramètre qui dépend de la taille du widget.

```
void Tetrahedron::paintGL()
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   draw();
}
```

La fonction `paintGL()` est invoquée dès que le widget doit être redessiné. Elle ressemble à `QWidget::paintEvent()`, mais des fonctions OpenGL remplacent les fonctions de `QPainter`. Le dessin est effectué par la fonction privée `draw()`.

```
void Tetrahedron::draw()
{
   static const GLfloat P1[3] = { 0.0, -1.0, +2.0 };
   static const GLfloat P2[3] = { +1.73205081, -1.0, -1.0 };
   static const GLfloat P3[3] = { -1.73205081, -1.0, -1.0 };
   static const GLfloat P4[3] = { 0.0, +2.0, 0.0 };
```

```
    static const GLfloat * const coords[4][3] = {
       { P1, P2, P3 }, { P1, P3, P4 }, { P1, P4, P2 }, { P2, P4, P3 }
    };

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -10.0);
    glRotatef(rotationX, 1.0, 0.0, 0.0);
    glRotatef(rotationY, 0.0, 1.0, 0.0);
    glRotatef(rotationZ, 0.0, 0.0, 1.0);

    for (int i = 0; i < 4; ++i) {
        glLoadName(i);
        glBegin(GL_TRIANGLES);
        qglColor(faceColors[i]);
        for (int j = 0; j < 3; ++j) {
           glVertex3f(coords[i][j][0], coords[i][j][1],
                      coords[i][j][2]);
        }
        glEnd();
    }
}
```

Dans `draw()`, nous dessinons le tétraèdre, en tenant compte des rotations *x*, *y* et *z* et des couleurs conservées dans le tableau `faceColors`. Tout est en code OpenGL standard, sauf l'appel de `qglColor()`. Nous aurions pu choisir plutôt une des fonctions OpenGL `glColor3d()` ou `glIndex()` en fonction du mode.

```
void Tetrahedron::mousePressEvent(QMouseEvent *event)
{
    lastPos = event->pos();
}

void Tetrahedron::mouseMoveEvent(QMouseEvent *event)
{
    GLfloat dx = GLfloat(event->x() - lastPos.x()) / width();
    GLfloat dy = GLfloat(event->y() - lastPos.y()) / height();

    if (event->buttons() & Qt::LeftButton) {
        rotationX += 180 * dy;
        rotationY += 180 * dx;
        updateGL();
    } else if (event->buttons() & Qt::RightButton) {
        rotationX += 180 * dy;
        rotationZ += 180 * dx;
        updateGL();
    }
    lastPos = event->pos();
}
```

Les fonctions `mousePressEvent()` et `mouseMoveEvent()` sont réimplémentées dans `QWidget` pour permettre à l'utilisateur de faire pivoter la vue en cliquant dessus et en la faisant glisser.

Le bouton gauche de la souris contrôle la rotation autour des axes *x* et *y*, et le bouton droit autour des axes *x* et *z*.

Après avoir modifié la variable `rotationX` et une des variables `rotationY` ou `rotationZ`, nous appelons `updateGL()` pour redessiner la scène.

```
void Tetrahedron::mouseDoubleClickEvent(QMouseEvent *event)
{
   int face = faceAtPosition(event->pos());
   if (face != -1) {
       QColor color = QColorDialog::getColor(faceColors[face], this);
       if (color.isValid()) {
           faceColors[face] = color;
           updateGL();
        }
    }
}
```

`mouseDoubleClickEvent()` est réimplémentée dans `QWidget` pour permettre à l'utilisateur de définir la couleur d'une des faces du tétraèdre en double-cliquant dessus. Nous invoquons la fonction privée `faceAtPosition()` pour déterminer quelle face se situe sous le curseur, s'il y en a une. Si l'utilisateur a double-cliqué sur une face, nous appelons `QColorDialog::getColor()` pour obtenir une nouvelle couleur pour cette face. Nous mettons ensuite à jour le tableau `faceColors` pour tenir compte de la nouvelle couleur et nous invoquons `updateGL()` pour redessiner la scène.

```
int Tetrahedron::faceAtPosition(const QPoint &pos)
{
   const int MaxSize = 512;
   GLuint buffer[MaxSize];
   GLint viewport[4];

   glGetIntegerv(GL_VIEWPORT, viewport);
   glSelectBuffer(MaxSize, buffer);
   glRenderMode(GL_SELECT);

   glInitNames();
   glPushName(0);

   glMatrixMode(GL_PROJECTION);
   glPushMatrix();
   glLoadIdentity();
   gluPickMatrix(GLdouble(pos.x()), GLdouble(viewport[3] - pos.y()),
                 5.0, 5.0, viewport);
   GLfloat x = GLfloat(width()) / height();
   glFrustum(-x, x, -1.0, 1.0, 4.0, 15.0);
   draw();
   glMatrixMode(GL_PROJECTION);
   glPopMatrix();
```

```
    if (!glRenderMode(GL_RENDER))
       return -1;
    return buffer[3];
}
```

La fonction `faceAtPosition()` retourne le nombre de faces à une certaine position sur le widget, ou –1 si aucune face ne se trouve à cet endroit. Le code OpenGL permettant de déterminer ceci est quelque peu complexe. En résumé, nous affichons la scène en mode `GL_SELECT` pour profiter des fonctionnalités de sélection d'OpenGL, puis nous récupérons le numéro de la face (son "nom") dans l'enregistrement du nombre d'accès d'OpenGL.

Voici `main.cpp` :

```
#include <QApplication>
#include <iostream>

#include "tetrahedron.h"

using namespace std;

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   if (!QGLFormat::hasOpenGL()) {
       cerr << "This system has no OpenGL support" << endl;
       return 1;
   }

   Tetrahedron tetrahedron;
   tetrahedron.setWindowTitle(QObject::tr("Tetrahedron"));
   tetrahedron.resize(300, 300);
   tetrahedron.show();

   return app.exec();
}
```

Si le système de l'utilisateur ne prend pas en charge OpenGL, nous imprimons un message d'erreur sur la console et nous retournons immédiatement.

Pour associer l'application au module *QtOpenGL* et à la bibliothèque OpenGL du système, le fichier `.pro` doit contenir cette entrée :

```
QT += opengl
```

L'application Tetrahedron est terminée. Pour plus d'informations sur le module *QtOpenGL*, consultez la documentation de référence de `QGLWidget`, `QGLFormat`, `QGLContext`, `QGLColormap` et `QGLPixelBuffer`.

9

Glisser-déposer

Au sommaire de ce chapitre

- ✔ Activer le glisser-déposer
- ✔ Prendre en charge les types personnalisés de glisser
- ✔ Gérer le presse-papiers

Le glisser-déposer est un moyen moderne et intuitif de transférer des informations dans une application ou entre différentes applications. Cette technique est souvent proposée en complément du support du presse-papiers pour déplacer et copier des données.

Dans ce chapitre, nous verrons comment ajouter la prise en charge du glisser-déposer à une application et comment gérer des formats personnalisés. Nous étudierons également la manière de réutiliser le code du glisser-déposer pour ajouter le support du presse-papiers. Cette réutilisation du code est possible parce que les deux mécanismes sont basés sur `QMimeData`, une classe capable de fournir des données dans divers formats.

Activer le glisser-déposer

Le glisser-déposer implique deux actions distinctes : glisser et déposer. On peut faire glisser et/ou déposer des éléments sur les widgets Qt.

Notre premier exemple vous présente comment faire accepter à une application Qt un glisser initié par une autre application. L'application Qt est une fenêtre principale avec un widget central `QTextEdit`. Quand l'utilisateur fait glisser un fichier texte du bureau ou de l'explorateur de fichiers vers l'application, celle-ci charge le fichier dans le `QTextEdit.`

Voici la définition de la classe `MainWindow` de notre exemple :

```
class MainWindow : public QMainWindow
{
   Q_OBJECT

public:
   MainWindow();

protected:
   void dragEnterEvent(QDragEnterEvent *event);
   void dropEvent(QDropEvent *event);

private:
   bool readFile(const QString &fileName);
   QTextEdit *textEdit;
};
```

La classe `MainWindow` réimplémente `dragEnterEvent()` et `dropEvent()` dans `QWidget`. Vu que l'objectif de notre exemple est de présenter le glisser-déposer, la majorité des fonctionnalités qu'une classe de fenêtre principale devrait contenir a été omise.

```
MainWindow::MainWindow()
{
   textEdit = new QTextEdit;
   setCentralWidget(textEdit);

   textEdit->setAcceptDrops(false);
   setAcceptDrops(true);

   setWindowTitle(tr("Text Editor"));
}
```

Dans le constructeur, nous créons un `QTextEdit` et nous le définissons comme widget central. Par défaut, `QTextEdit` accepte des glisser sous forme de texte provenant d'autres applications, et si l'utilisateur y dépose un fichier, le nom de fichier sera intégré dans le texte. Les événements `drop` étant transmis de l'enfant au parent, nous obtenons les événements `drop` pour toute la fenêtre dans `MainWindow` en désactivant le déposer dans `QTextEdit` et en l'activant dans la fenêtre principale.

```
void MainWindow::dragEnterEvent(QDragEnterEvent *event)
{
   if (event->mimeData()->hasFormat("text/uri-list"))
       event->acceptProposedAction();
}
```

`dragEnterEvent()` est appelée dès que l'utilisateur fait glisser un objet sur un widget. Si nous invoquons `acceptProposedAction()` sur l'événement, nous indiquons que l'utilisateur est en mesure de déposer cet objet sur ce widget. Par défaut, le widget n'accepterait pas le glisser. Qt modifie automatiquement le pointeur pour signaler à l'utilisateur si le widget est en mesure d'accepter le dépôt.

Dans notre cas, nous voulons que l'utilisateur puisse faire glisser des fichiers, mais rien d'autre. Pour ce faire, nous vérifions le type MIME du glisser. Le type MIME `text/uri-list` est utilisé pour stocker une liste d'URI (universal resource identi?er), qui peuvent être des noms de fichiers, des URL (comme des chemins d'accès HTTP ou FTP) ou d'autres identifiants globaux de ressource. Les types MIME standards sont définis par l'IANA (*Internet Assigned Numbers Authority*). Ils sont constitués d'un type et d'un sous-type séparés par un slash. Les types MIME sont employés par le presse-papiers et par le système du glisser-déposer pour identifier les différents types de données. La liste officielle des types MIME est disponible à l'adresse suivante : **http://www.iana.org/assignments/media-types/.**

```
void MainWindow::dropEvent(QDropEvent *event)
{
   QList<QUrl> urls = event->mimeData()->urls();
   if (urls.isEmpty())
       return;

   QString fileName = urls.first().toLocalFile();
   if (fileName.isEmpty())
       return;

   if (readFile(fileName))
       setWindowTitle(tr("%1 - %2").arg(fileName)
                                   .arg(tr("Drag File")));
}
```

`dropEvent()` est appelée dès que l'utilisateur dépose un objet sur un widget. Nous appelons `QMimeData::urls()` pour obtenir une liste des `QUrl`. En général, les utilisateurs ne font glisser qu'un seul fichier à la fois, mais il est possible d'en faire glisser plusieurs en même temps grâce à une sélection. S'il y a plusieurs URL ou si l'URL ne correspond pas à un nom de fichier local, nous retournons immédiatement.

`QWidget` propose aussi `dragMoveEvent()` et `dragLeaveEvent()`, mais ces fonctions n'ont généralement pas besoin d'être réimplémentées.

Le deuxième exemple illustre la façon d'initier un glisser et d'accepter un déposer. Nous allons créer une sous-classe `QListWidget` qui prend en charge le glisser-déposer et

nous l'utiliserons en tant que composant de l'application Project Chooser présentée en Figure 9.1.

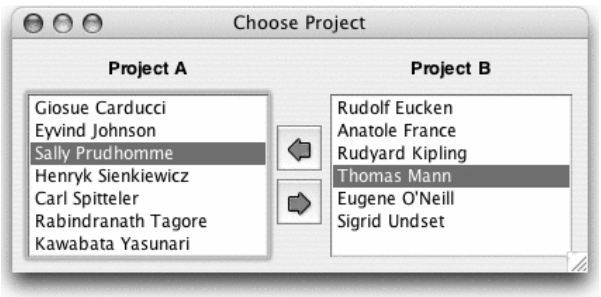


Figure 9.1
L'application Project Chooser

L'application Project Chooser présente à l'utilisateur deux widgets liste remplis de noms. Chaque widget représente un projet. L'utilisateur peut faire glisser et déposer les noms dans les widgets liste pour déplacer une personne d'un projet à un autre.

Le code du glisser-déposer se situe en globalité dans la sous-classe `QListWidget`. Voici la définition de classe :

```
class ProjectListWidget : public QListWidget
{
   Q_OBJECT

public:
   ProjectListWidget(QWidget *parent = 0);

protected:
   void mousePressEvent(QMouseEvent *event);
   void mouseMoveEvent(QMouseEvent *event);
   void dragEnterEvent(QDragEnterEvent *event);
   void dragMoveEvent(QDragMoveEvent *event);
   void dropEvent(QDropEvent *event);

private:
   void startDrag();

   QPoint startPos;
};
```

La classe `ProjectListWidget` réimplémente cinq gestionnaires d'événements déclarés dans `QWidget`.

```
ProjectListWidget::ProjectListWidget(QWidget *parent)
   : QListWidget(parent)
{
   setAcceptDrops(true);
}
```

Dans le constructeur, nous activons le déposer sur le widget liste.

```
void ProjectListWidget::mousePressEvent(QMouseEvent *event)
{
   if (event->button() == Qt::LeftButton)
      startPos = event->pos();
   QListWidget::mousePressEvent(event);
}
```

Quand l'utilisateur appuie sur le bouton gauche de la souris, nous stockons l'emplacement de cette dernière dans la variable privée `startPos`. Nous appelons l'implémentation de `mousePressEvent()` du `QListWidget` pour nous assurer que ce dernier a la possibilité de traiter des événements "bouton souris enfoncé" comme d'habitude.

```
void ProjectListWidget::mouseMoveEvent(QMouseEvent *event)
{
   if (event->buttons() & Qt::LeftButton) {
      int distance = (event->pos() - startPos).manhattanLength();
      if (distance >= QApplication::startDragDistance())
         startDrag();
   }
   QListWidget::mouseMoveEvent(event);
}
```

Quand l'utilisateur déplace le pointeur tout en maintenant le bouton gauche de la souris enfoncé, nous commençons un glisser. Nous calculons la distance entre la position actuelle de la souris et la position où le bouton gauche a été enfoncé. Si la distance est supérieure à la distance recommandée pour démarrer un glisser de `QApplication` (normalement 4 pixels), nous appelons la fonction privée `startDrag()` pour débuter le glisser. Ceci évite d'initier un glisser si la main de l'utilisateur a tremblé.

```
void ProjectListWidget::startDrag()
{
   QListWidgetItem *item = currentItem();
   if (item) {
      QMimeData *mimeData = new QMimeData;
      mimeData->setText(item->text());
      QDrag *drag = new QDrag(this);
      drag->setMimeData(mimeData);
      drag->setPixmap(QPixmap(":/images/person.png"));
      if (drag->start(Qt::MoveAction) == Qt::MoveAction)
         delete item;
   }
}
```

Dans `startDrag()`, nous créons un objet de type `QDrag`, `this` étant son parent. L'objet `QDrag` enregistre les données dans un objet `QMimeData`. Dans notre exemple, nous fournissons les données sous forme de chaîne `text/plain` au moyen de `QMimeData::setText()`. `QMimeData` propose plusieurs fonctions permettant de gérer les types les plus courants de glisser (images, URL, couleurs, etc.) et peut gérer des types MIME arbitraires représentés comme

étant des `QByteArray`. L'appel de `QDrag::setPixmap()` définit l'icône qui suit le pointeur pendant le glisser.

L'appel de `QDrag::start()` débute le glisser et se bloque jusqu'à ce que l'utilisateur dépose ou annule le glisser. Elle reçoit en argument une combinaison des glisser pris en charge (`Qt::CopyAction`, `Qt::MoveAction` et `Qt::LinkAction`) et retourne le glisser qui a été exécuté (ou `Qt::IgnoreAction` si aucun glisser n'a été exécuté). L'action exécutée dépend de ce que le widget source autorise, de ce que la cible supporte et des touches de modification enfoncées au moment de déposer. Après l'appel de `start()`, Qt prend possession de l'objet glissé et le supprimera quand il ne sera plus nécessaire.

```
void ProjectListWidget::dragEnterEvent(QDragEnterEvent *event)
{
   ProjectListWidget *source =
         qobject_cast<ProjectListWidget *>(event->source());
   if (source && source != this) {
       event->setDropAction(Qt::MoveAction);
       event->accept();
   }
}
```

Le widget `ProjectListWidget` ne sert pas uniquement à initialiser des glisser, il accepte aussi des glisser provenant d'un autre `ProjectListWidget` de la même application. `QDragEnterEvent::source()` retourne un pointeur vers le widget à l'origine du glisser si ce widget fait partie de la même application ; sinon elle renvoie un pointeur nul. Nous utilisons `qobject_cast<T>()` pour nous assurer que le glisser provient d'un `ProjectListWidget`. Si tout est correct, nous informons Qt que nous sommes prêts à accepter l'action en tant qu'action de déplacement.

```
void ProjectListWidget::dragMoveEvent(QDragMoveEvent *event)
{
   ProjectListWidget *source =
         qobject_cast<ProjectListWidget *>(event->source());
   if (source && source != this) {
       event->setDropAction(Qt::MoveAction);
       event->accept();
   }
}
```

Le code dans `dragMoveEvent()` est identique à ce que nous effectué dans `dragEnterEvent()`. Il est nécessaire parce que nous devons remplacer l'implémentation de la fonction dans `QListWidget` (en fait dans `QAbstractItemView`).

```
void ProjectListWidget::dropEvent(QDropEvent *event)
{
   ProjectListWidget *source =
         qobject_cast<ProjectListWidget *>(event->source());
   if (source && source != this) {
       addItem(event->mimeData()->text());
       event->setDropAction(Qt::MoveAction);
```

```
            event->accept();
        }
    }
```

Dans `dropEvent()`, nous récupérons le texte glissé à l'aide de `QMimeData::text()` et nous créons un élément avec ce texte. Nous avons également besoin d'accepter l'événement comme étant une "action de déplacement" afin de signaler au widget source qu'il peut maintenant supprimer la version originale de l'élément glissé.

Le glisser-déposer est un mécanisme puissant permettant de transférer des données entre des applications. Cependant, dans certains cas, il est possible d'implémenter le glisser-déposer sans utiliser les fonctionnalités de glisser-déposer de Qt. Si tout ce que nous souhaitons se limite à déplacer des données dans un widget d'une application, il suffit de réimplémenter `mousePressEvent()` et `mouseReleaseEvent()`.

Prendre en charge les types personnalisés de glisser

Jusqu'à présent, nous nous sommes basés sur la prise en charge de `QMimeData` des types MIME communs. Nous avons donc appelé `QMimeData::setText()` pour créer un glisser de texte et nous avons exécuté `QMimeData:urls()` pour récupérer le contenu d'un glisser `text/uri-list`. Si nous voulons faire glisser du texte brut, du texte HTML, des images, des URL ou des couleurs, nous pouvons employer `QMimeData` sans formalité. Mais si nous souhaitons faire glisser des données personnalisées, nous devons faire un choix entre plusieurs possibilités :

1. Nous pouvons fournir des données arbitraires sous forme de `QByteArray` en utilisant `QMimeData::setData()` et les extraire ultérieurement avec `QMimeData::data()`.
2. Nous pouvons dériver `QMimeData` et réimplémenter `formats()` et `retrieveData()` pour gérer nos types personnalisés de données.
3. S'agissant du glisser-déposer dans une seule application, nous avons la possibilité de dériver `QMimeData` et de stocker les données dans la structure de notre choix.

La première approche n'implique pas de dérivation, mais présente certains inconvénients : nous devons convertir notre structure de données en `QByteArray` même si le glisser n'est pas accepté à la fin, et si nous voulons proposer plusieurs types MIME pour interagir correctement avec une large gamme d'applications, nous devons enregistrer les données plusieurs fois (une fois pour chaque type MIME). Si les données sont nombreuses, l'application peut être ralentie inutilement. Les deuxième et troisième approches permettent d'éviter ou de minimiser ces problèmes. Ainsi, nous avons un contrôle total et nous pouvons les utiliser ensemble.

Pour vous présenter le fonctionnement de ces approches, nous vous montrerons comment ajouter des fonctions de glisser-déposer à un `QTableWidget`. Le glisser prendra en charge les types

MIME suivants : `text/plain`, `text/html` et `text/csv`. En utilisant la première approche, voici comment débute un glisser :

```
void MyTableWidget::mouseMoveEvent(QMouseEvent *event)
{
   if (event->buttons() & Qt::LeftButton) {
      int distance = (event->pos() - startPos).manhattanLength();
      if (distance >= QApplication::startDragDistance())
          startDrag();
   }
   QTableWidget::mouseMoveEvent(event);
}

void MyTableWidget::startDrag()
{
   QString plainText = selectionAsPlainText();
   if (plainText.isEmpty())
       return;

   QMimeData *mimeData = new QMimeData;
   mimeData->setText(plainText);
   mimeData->setHtml(toHtml(plainText));
   mimeData->setData("text/csv", toCsv(plainText).toUtf8());

   QDrag *drag = new QDrag(this);
   drag->setMimeData(mimeData);
   if (drag->start(Qt::CopyAction | Qt::MoveAction) == Qt::MoveAction)
       deleteSelection();
}
```

La fonction privée `startDrag()` a été invoquée dans `mouseMoveEvent()` pour commencer à faire glisser une sélection rectangulaire. Nous définissons les types MIME `text/plain` et `text/html` avec `setText()` et `setHtml()` et nous configurons le type `text/csv` avec `setData()`, qui reçoit un type MIME arbitraire et un `QByteArray`. Le code de `selectionAsString()` est plus ou moins le même que la fonction `Spreadsheet::copy()` du Chapitre 4.

```
QString MyTableWidget::toCsv(const QString &plainText)
{
   QString result = plainText;
   result.replace("\\", "\\\\");
   result.replace("\"", "\\\"");
   result.replace("\t", "\", \"");
   result.replace("\n", "\"\n\"");
   result.prepend("\"");
   result.append("\"");
   return result;
}

QString MyTableWidget::toHtml(const QString &plainText)
{
   QString result = Qt::escape(plainText);
   result.replace("\t", "<td>");
```

```
    result.replace("\n", "\n<tr><td>");
    result.prepend("<table>\n<tr><td>");
    result.append("\n</table>");
    return result;
}
```

Les fonctions `toCsv()` et `toHtml()` convertissent une chaîne "tabulations et sauts de ligne" en une chaîne CSV (comma-separated values, valeurs séparées par des virgules) ou HTML. Par exemple, les données

```
Red     Green     Blue
Cyan    Yellow    Magenta
```

sont converties en

```
"Red",    "Green",    "Blue"
"Cyan",   "Yellow",   "Magenta"
```

ou en

```
<table>
<tr><td>Red<td>Green<td>Blue
<tr><td>Cyan<td>Yellow<td>Magenta
</table>
```

La conversion est effectuée de la manière la plus simple possible, en exécutant `QString::replace()`. Pour éviter les caractères spéciaux HTML, nous employons `Qt::escape()`.

```
void MyTableWidget::dropEvent(QDropEvent *event)
{
    if (event->mimeData()->hasFormat("text/csv")) {
        QByteArray csvData = event->mimeData()->data("text/csv");
        QString csvText = QString::fromUtf8(csvData);
        ...
        event->acceptProposedAction();
    } else if (event->mimeData()->hasFormat("text/plain")) {
        QString plainText = event->mimeData()->text();
        ...
        event->acceptProposedAction();
    }
}
```

Même si nous fournissons les données dans trois formats différents, nous n'acceptons que deux d'entre eux dans `dropEvent()`. Si l'utilisateur fait glisser des cellules depuis un `QTableWidget` vers un éditeur HTML, nous voulons que les cellules soient converties en un tableau HTML. Mais si l'utilisateur fait glisser un code HTML arbitraire vers un `QTableWidget`, nous ne voulons pas l'accepter.

Pour que cet exemple fonctionne, nous devons également appeler `setAcceptDrops(true)` et `setSelectionMode(ContiguousSelection)` dans le constructeur de `MyTableWidget`.

Nous allons refaire notre exemple, mais cette fois-ci nous dériverons `QMimeData` pour ajourner ou éviter les conversions (potentiellement onéreuses en termes de performances) entre `QTableWidgetItem` et `QByteArray`. Voici la définition de notre sous-classe :

```
class TableMimeData : public QMimeData
{
   Q_OBJECT

public:
   TableMimeData(const QTableWidget *tableWidget,
                 const QTableWidgetSelectionRange &range);

   const QTableWidget *tableWidget() const { return myTableWidget; }
   QTableWidgetSelectionRange range() const { return myRange; }
   QStringList formats() const;

protected:
   QVariant retrieveData(const QString &format,
                         QVariant::Type preferredType) const;

private:
   static QString toHtml(const QString &plainText);
   static QString toCsv(const QString &plainText);

   QString text(int row, int column) const;
   QString rangeAsPlainText() const;

   const QTableWidget *myTableWidget;
   QTableWidgetSelectionRange myRange;
   QStringList myFormats;
};
```

Au lieu de stocker les données réelles, nous enregistrons un `QTableWidgetSelectionRange` qui spécifie quelles cellules ont été glissées et nous conservons un pointeur vers `QTableWidget`. Les fonctions `formats()` et `retrieveData()` sont réimplémentées dans `QMimeData`.

```
TableMimeData::TableMimeData(const QTableWidget *tableWidget,
                             const QTableWidgetSelectionRange &range)
{
   myTableWidget = tableWidget;
   myRange = range;
   myFormats << "text/csv" << "text/html" << "text/plain";
}
```

Dans le constructeur, nous initialisons les variables privées.

```
QStringList TableMimeData::formats() const
{
   return myFormats;
}
```

La fonction `formats()` retourne une liste de types MIME fournie par l'objet MIME. L'ordre précis des formats n'est généralement pas important, mais il est recommandé de placer les "meilleurs" formats en premier. Il arrive en effet que les applications qui prennent en charge de nombreux formats choisissent le premier qui convient.

```
QVariant TableMimeData::retrieveData(const QString &format,
                                     QVariant::Type preferredType) const
{
   if (format == "text/plain") {
       return rangeAsPlainText();
   } else if (format == "text/csv") {
       return toCsv(rangeAsPlainText());
   } else if (format == "text/html") {
       return toHtml(rangeAsPlainText());
   } else {
       return QMimeData::retrieveData(format, preferredType);
   }
}
```

La fonction `retrieveData()` retourne les données d'un type MIME particulier sous forme de `QVariant`. La valeur du paramètre de format correspond normalement à une des chaînes retournées par `formats()`, mais nous ne pouvons pas en attester, étant donné que toutes les applications ne comparent pas le type MIME à `formats()`. Les fonctions d'accès `text()`, `html()`, `urls()`, `imageData()`, `colorData()` et `data()` proposées par `QMimeData` sont implémentées en termes de `retrieveData()`.

Le paramètre `preferredType` est un bon indicateur du type que nous devrions placer dans `QVariant`. Ici, nous l'ignorons et nous faisons confiance à `QMimeData` pour convertir la valeur de retour dans le type souhaité, si nécessaire.

```
void MyTableWidget::dropEvent(QDropEvent *event)
{
   const TableMimeData *tableData =
         qobject_cast<const TableMimeData *>(event->mimeData());

   if (tableData) {
       const QTableWidget *otherTable = tableData->tableWidget();
       QTableWidgetSelectionRange otherRange = tableData->range();
       ...
       event->acceptProposedAction();
   } else if (event->mimeData()->hasFormat("text/csv")) {
       QByteArray csvData = event->mimeData()->data("text/csv");
       QString csvText = QString::fromUtf8(csvData);
       ...
       event->acceptProposedAction();
   } else if (event->mimeData()->hasFormat("text/plain")) {
       QString plainText = event->mimeData()->text();
       ...
       event->acceptProposedAction();
    }
    QTableWidget::mouseMoveEvent(event);
}
```

La fonction `dropEvent()` ressemble à celle présentée précédemment dans cette section, mais cette fois-ci nous l'optimisons en vérifiant d'abord si nous pouvons convertir en toute sécurité l'objet `QMimeData` en `TableMimeData`. Si `qobject_cast<T>()` fonctionne, cela signifie qu'un `MyTableWidget` de la même application était à l'origine du glisser, et nous pouvons directement accéder aux données de la table au lieu de passer par l'API de `QMimeData`. Si la conversion échoue, nous extrayons les données de façon habituelle.

Dans cet exemple, nous avons codé le texte CSV avec le format de codage UTF-8. Si nous voulions être sûrs d'utiliser le bon codage, nous aurions pu utiliser le paramètre `charset` du type MIME `text/plain` de sorte à spécifier un codage explicite. Voici quelques exemples :

```
text/plain;charset=US-ASCII
text/plain;charset=ISO-8859-1
text/plain;charset=Shift_JIS
text/plain;charset=UTF-8
```

Gérer le presse-papiers

La plupart des applications emploient la gestion intégrée du presse-papiers de Qt d'une manière ou d'une autre. Par exemple, la classe `QTextEdit` propose les slots `cut()`, `copy()` et `paste()`, de même que des raccourcis clavier. Vous n'avez donc pas besoin de code supplémentaire, ou alors très peu.

Quand vous écrivez vos propres classes, vous pouvez accéder au presse-papiers par le biais de `QApplication::clipboard()`, qui retourne un pointeur vers l'objet `QClipboard` de l'application. Gérer le presse-papiers s'avère assez facile : vous appelez `setText()`, `setImage()` ou `setPixmap()` pour placer des données dans le presse-papiers, puis vous appelez `text()`, `image()` ou `pixmap()` pour y récupérer les données. Nous avons déjà analysé des exemples d'utilisation du presse-papiers dans l'application Spreadsheet du Chapitre 4.

Pour certaines applications, la fonctionnalité intégrée peut être insuffisante. Par exemple, nous voulons pouvoir fournir des données qui ne soient pas uniquement du texte ou une image, ou proposer des données en différents formats pour un maximum d'interopérabilité avec d'autres applications. Ce problème ressemble beaucoup à ce que nous avons rencontré précédemment avec le glisser-déposer et la réponse est similaire : nous pouvons dériver `QMimeData` et réimplémenter quelques fonctions virtuelles.

Si notre application prend en charge le glisser-déposer *via* une sous-classe `QMimeData` personnalisée, nous pouvons simplement réutiliser cette sous-classe et la placer dans le presse-papiers en employant la fonction `setMimeData()`. Pour récupérer les données, nous avons la possibilité d'invoquer `mimeData()` sur le presse-papiers.

Sous X11, il est habituellement possible de coller une sélection en cliquant sur le bouton du milieu d'une souris dotée de trois boutons. Vous faites alors appel à un presse-papiers de "sélection" distinct. Si vous souhaitez que vos widgets supportent ce genre de presse-papiers en complément du presse-papiers standard, vous devez transmettre `QClipboard::Selection`

comme argument supplémentaire aux divers appels du presse-papiers. Voici par exemple comment nous réimplémenterions `mouseReleaseEvent()` dans un éditeur de texte pour prendre en charge le collage avec le bouton du milieu de la souris :

```
void MyTextEditor::mouseReleaseEvent(QMouseEvent *event)
{
   QClipboard *clipboard = QApplication::clipboard();
   if (event->button() == Qt::MidButton
         && clipboard->supportsSelection()) {
      QString text = clipboard->text(QClipboard::Selection);
      pasteText(text);
   }
}
```

Sous X11, la fonction `supportsSelection()` retourne `true`. Sur les autres plates-formes, elle renvoie `false`.

Si vous voulez être informé dès que le contenu du presse-papiers change, vous pouvez connecter le signal `QClipboard::dataChanged()` à un slot personnalisé.

10

Classes d'affichage d'éléments

Au sommaire de ce chapitre

- ✔ Utiliser les classes dédiées à l'affichage d'éléments
- ✔ Utiliser des modèles prédéfinis
- ✔ Implémenter des modèles personnalisés
- ✔ Implémenter des délégués personnalisés

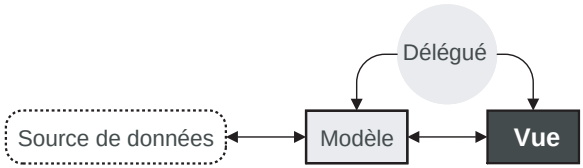
Beaucoup d'applications ont pour objectif la recherche, l'affichage et la modification d'éléments individuels appartenant à un ensemble de données. Ces données peuvent se trouver dans des fichiers, des bases de données ou des serveurs de réseau. L'approche standard relative au traitement des ensembles de données consiste à utiliser les classes d'affichage d'éléments de Qt.

Dans les versions antérieures de Qt, les widgets d'affichage d'éléments étaient alimentés avec la totalité de l'ensemble de données ; les utilisateurs pouvaient effectuer toutes leurs recherches et modifications sur les données hébergées dans le widget, et à un

moment donné, les modifications étaient sauvegardées dans la source de données. Même si elle est simple à comprendre et à utiliser, cette approche n'est pas adaptée aux très grands ensembles de données, ni pour l'affichage du même ensemble de données dans deux ou plusieurs widgets différents.

Le langage Smalltalk a fait connaître une méthode flexible permettant de visualiser de grands ensembles de données : MVC (Modèle-Vue-Contrôleur). Dans l'approche MVC, le *modèle* représente l'ensemble de données et il se charge de récupérer les données nécessaires pour afficher et enregistrer toute modification. Chaque type d'ensemble de données possède son propre modèle, mais l'API que les modèles proposent aux vues est identique quel que soit l'ensemble de données sous-jacent. La *vue* présente les données à l'utilisateur. Seule une quantité limitée de données d'un grand ensemble sera visible en même temps, c'est-à-dire celles demandées par la vue. Le *contrôleur* sert d'intermédiaire entre l'utilisateur et la vue ; il convertit les actions utilisateur en requêtes pour rechercher ou modifier des données, que la vue transmet ensuite au modèle si nécessaire.

Figure 10.1
L'architecture modèle/vue de Qt



Qt propose une architecture modèle/vue inspirée de l'approche MVC (voir Figure 10.1). Dans Qt, le modèle se comporte de la même manière que pour le MVC classique. Mais à la place du contrôleur, Qt utilise une notion légèrement différente : le *délégué*. Le délégué offre un contrôle précis de la manière dont les éléments sont affichés et modifiés. Qt fournit un délégué par défaut pour chaque type de vue. C'est suffisant pour la plupart des applications, c'est pourquoi nous n'avons généralement pas besoin de nous en préoccuper.

Grâce à l'architecture modèle/vue de Qt, nous avons la possibilité d'utiliser des modèles qui ne récupèrent que les données nécessaires à l'affichage de la vue. Nous gérons donc de très grands ensembles de données beaucoup plus rapidement et nous consommons moins de mémoire que si nous devions lire toutes les données. De plus, en enregistrant un modèle avec deux vues ou plus, nous donnons l'opportunité à l'utilisateur d'afficher et d'interagir avec les données de différentes manières, avec peu de surcharge (voir Figure 10.2). Qt synchronise automatiquement plusieurs vues, reflétant les changements apportés dans l'une d'elles dans toutes les autres. L'architecture modèle/vue présente un autre avantage : si nous décidons de modifier la façon dont l'ensemble de données sous-jacent est enregistré, nous n'avons qu'à changer le modèle ; les vues continueront à se comporter correctement.

En général, nous ne devons présenter qu'un nombre relativement faible d'éléments à l'utilisateur. Dans ces cas fréquents, nous pouvons utiliser les classes d'affichage d'éléments de Qt (`QListWidget`, `QTableWidget` et `QTreeWidget`) spécialement conçues à cet effet et directement y enregistrer des éléments. Ces classes se comportent de manière similaire aux

classes d'affichage d'éléments proposées par les versions antérieures de Qt. Elles stockent leurs données dans des "éléments" (par exemple, un `QTableWidget` contient des `QTableWidgetItem`). En interne, ces classes s'appuient sur des modèles personnalisés qui affichent les éléments destinés aux vues.

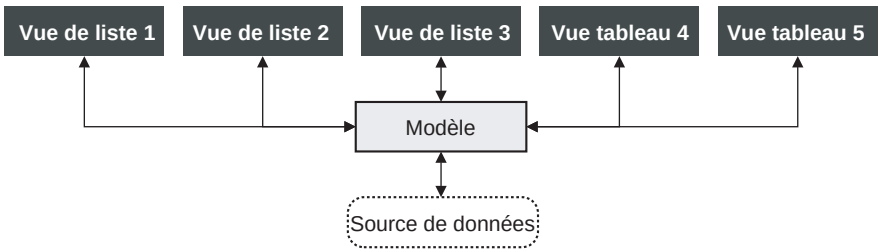


Figure 10.2
Un modèle peut desservir plusieurs vues

S'agissant des grands ensembles de données, la duplication des données est souvent peu recommandée. Dans ces cas, nous pouvons utiliser les vues de Qt (`QListView`, `QTableView`, et `QTreeView`,), en association avec un modèle de données, qui peut être un modèle personnalisé ou un des modèles prédéfinis de Qt. Par exemple, si l'ensemble de données se trouve dans une base de données, nous pouvons combiner un `QTableView` avec un `QSqlTableModel`.

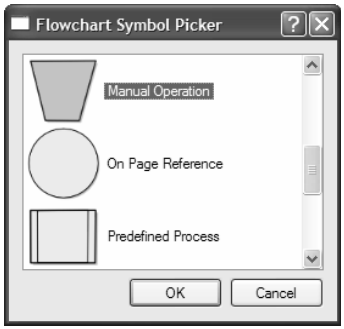
Utiliser les classes dédiées à l'affichage d'éléments

Utiliser les sous-classes d'affichage d'éléments de Qt est généralement plus simple que de définir un modèle personnalisé, et cette solution est plus appropriée quand la séparation du modèle et de la vue ne présente aucun intérêt particulier. Nous avons employé cette technique dans le Chapitre 4 quand nous avons dérivé `QTableWidget` et `QTableWidgetItem` pour implémenter la fonctionnalité de feuille de calcul.

Dans cette section, nous verrons comment utiliser les sous-classes d'affichage d'éléments pour afficher des éléments. Le premier exemple vous présente un `QListWidget` en lecture seule, le deuxième exemple vous montre un `QTableWidget` modifiable et le troisième vous expose un `QTreeWidget` en lecture seule.

Nous commençons par une boîte de dialogue simple qui propose à l'utilisateur de choisir un symbole d'organigramme dans une liste, comme illustré en Figure 10.3. Chaque élément est composé d'une icône, de texte et d'un ID unique.

Figure 10.3
L'application Flowchart Symbol Picker



Analysons d'abord un extrait du fichier d'en-tête de la boîte de dialogue :

```
class FlowChartSymbolPicker : public QDialog
{
   Q_OBJECT

public:
   FlowChartSymbolPicker(const QMap<int, QString> &symbolMap,
                         QWidget *parent = 0);

   int selectedId() const { return id; }
   void done(int result);
   ...
};
```

Quand nous construisons la boîte de dialogue, nous devons lui transmettre un `QMap<int,QString>`, et après son exécution, nous pouvons récupérer l'ID choisi (ou –1 si l'utilisateur n'a choisi aucun élément) en appelant `selectedId()`.

```
FlowChartSymbolPicker::FlowChartSymbolPicker(
      const QMap<int, QString> &symbolMap, QWidget *parent)
   : QDialog(parent)
{
   id = -1;

   listWidget = new QListWidget;
   listWidget->setIconSize(QSize(60, 60));

   QMapIterator<int, QString> i(symbolMap);

   while (i.hasNext()) {
      i.next();
      QListWidgetItem *item = new QListWidgetItem(i.value(),
                                                  listWidget);
      item->setIcon(iconForSymbol(i.value()));
      item->setData(Qt::UserRole, i.key());
   }
   ...
}
```

Nous initialisons `id` (le dernier ID sélectionné) à –1. Puis nous construisons un `QListWidget`, un widget dédié à l'affichage d'éléments. Nous parcourons chaque élément dans la liste des symboles d'organigramme et nous créons un `QListWidgetItem` pour représenter chacun d'eux. Le constructeur de `QListWidgetItem` reçoit un `QString` qui représente le texte à afficher, suivi par le parent `QListWidget`.

Nous définissons ensuite l'icône de l'élément et nous invoquons `setData()` pour enregistrer notre ID arbitraire dans le `QListWidgetItem`. La fonction privée `iconForSymbol()` retourne un `QIcon` pour un nom de symbole donné.

Les `QListWidgetItem` endossent plusieurs rôles, chacun ayant un `QVariant` associé. Les rôles les plus courants sont `Qt::DisplayRole`, `Qt::EditRole` et `Qt::IconRole`, pour lesquels il existe des fonctions dédiées d'accès et de réglage (`setText()`, `setIcon()`). Toutefois, il existe plusieurs autres rôles. Nous pouvons aussi définir des rôles personnalisés en spécifiant une valeur numérique de `Qt::UserRole` ou plus haut. Dans notre exemple, nous utilisons `Qt::UserRole` pour stocker l'ID de chaque élément.

La partie non représentée du constructeur se charge de créer les boutons, de disposer les widgets et de définir le titre de la fenêtre.

```
void FlowChartSymbolPicker::done(int result)
{
   id = -1;
   if (result == QDialog::Accepted) {
       QListWidgetItem *item = listWidget->currentItem();
       if (item)
          id = item->data(Qt::UserRole).toInt();
   }
   QDialog::done(result);
}
```

La fonction `done()` est réimplémentée dans `QDialog`. Elle est appelée quand l'utilisateur appuie sur OK ou Cancel. Si l'utilisateur a cliqué sur OK, nous récupérons l'élément pertinent et nous extrayons l'ID grâce à la fonction `data()`. Si nous étions intéressés par le texte de l'élément, nous aurions pu le récupérer en invoquant `item->data(Qt::DisplayRole).toString()` ou, ce qui est plus pratique, `item->text()`.

Par défaut, `QListWidget` est en lecture seule. Si nous voulions que l'utilisateur puisse modifier les éléments, nous aurions pu définir les déclencheurs de modification de la vue au moyen de `QAbstractItemView::setEditTriggers()` ; par exemple, configurer `QAbstractItemView::AnyKeyPressed` signifie que l'utilisateur peut modifier un élément simplement en commençant à taper quelque chose. Nous aurions aussi pu proposer un bouton Edit (ou peut-être des boutons Add et Delete) et les connecter aux slots, de sorte d'être en mesure de gérer les opérations de modification par programme.

Maintenant que nous avons vu comment utiliser une classe dédiée à l'affichage d'éléments pour afficher et sélectionner des données, nous allons étudier un exemple où nous pouvons modifier des données. Nous utilisons à nouveau une boîte de dialogue, mais cette fois-ci, elle présente un ensemble de coordonnées (x, y) que l'utilisateur peut modifier (voir Figure 10.4).

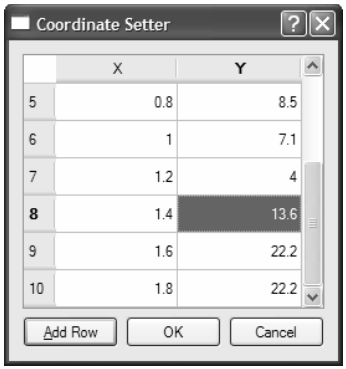


Figure 10.4
L'application Coordinate Setter

Comme pour l'exemple précédent, nous nous concentrerons sur le code d'affichage de l'élément, en commençant par le constructeur.

```
CoordinateSetter::CoordinateSetter(QList<QPointF> *coords,
                                   QWidget *parent)
   : QDialog(parent)
{
   coordinates = coords;

   tableWidget = new QTableWidget(0, 2);
   tableWidget->setHorizontalHeaderLabels(
         QStringList() << tr("X") << tr("Y"));

   for (int row = 0; row < coordinates->count(); ++row) {
      QPointF point = coordinates->at(row);
      addRow();
      tableWidget->item(row, 0)->setText(QString::number(point.x()));
      tableWidget->item(row, 1)->setText(QString::number(point.y()));
   }
   ...
}
```

Le constructeur de `QTableWidget` reçoit le nombre initial de lignes et de colonnes du tableau à afficher. Chaque élément dans un `QTableWidget` est représenté par un `QTableWidgetItem`, y compris les en-têtes horizontaux et verticaux. La fonction `setHorizontalHeaderLabels()` inscrit dans chaque élément horizontal du widget tableau le texte qui lui est fourni sous forme d'une liste de chaînes en argument. Par défaut, `QTableWidget` propose un en-tête vertical avec des lignes intitulées à partir de 1, ce qui correspond exactement à ce que nous recherchons, nous ne sommes donc pas contraints de configurer manuellement les intitulés de l'en-tête vertical.

Une fois que nous avons créé et centré les intitulés des colonnes, nous parcourons les coordonnées transmises. Pour chaque paire (*x*, *y*), nous créons deux `QTableWidgetItem` correspondant aux coordonnées *x* et *y*. Les éléments sont ajoutés au tableau grâce à `QTableWidget::setItem()`, qui reçoit une ligne et une colonne en plus de l'élément.

Par défaut, `QTableWidget` autorise les modifications. L'utilisateur peut modifier toute cellule du tableau en la recherchant puis en appuyant sur F2 ou simplement en saisissant quelque chose. Tous les changements effectués par l'utilisateur dans la vue se reflèteront automatiquement dans les `QTableWidgetItem`. Pour éviter les modifications, nous avons la possibilité d'appeler `setEditTriggers(QAbstractItemView::NoEditTriggers)`.

```
void CoordinateSetter::addRow()
{
   int row = tableWidget->rowCount();

   tableWidget->insertRow(row);

   QTableWidgetItem *item0 = new QTableWidgetItem;
   item0->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
   tableWidget->setItem(row, 0, item0);

   QTableWidgetItem *item1 = new QTableWidgetItem;
   item1->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
   tableWidget->setItem(row, 1, item1);

   tableWidget->setCurrentItem(item0);
}
```

Le slot `addRow()` est appelé lorsque l'utilisateur clique sur le bouton Add Row. Nous ajoutons une nouvelle ligne à l'aide de `insertRow()`. Si l'utilisateur essaie de modifier une cellule dans la nouvelle ligne, `QTableWidget` créera automatiquement un nouveau `QTableWidgetItem`.

```
void CoordinateSetter::done(int result)
{
   if (result == QDialog::Accepted) {
      coordinates->clear();
      for (int row = 0; row < tableWidget->rowCount(); ++row) {
          double x = tableWidget->item(row, 0)->text().toDouble();
          double y = tableWidget->item(row, 1)->text().toDouble();
          coordinates->append(QPointF(x, y));
      }
   }
   QDialog::done(result);
}
```

Enfin, quand l'utilisateur clique sur OK, nous effaçons les coordonnées qui avaient été transmises à la boîte de dialogue et nous créons un nouvel ensemble basé sur les coordonnées des éléments du `QTableWidget`.

Pour notre troisième et dernier exemple de widget dédié à l'affichage d'éléments de Qt, nous allons analyser quelques extraits de code d'une application qui affiche les paramètres d'une application Qt grâce à un `QTreeWidget` (voir Figure 10.5). La lecture seule est l'option par défaut de `QTreeWidget`.

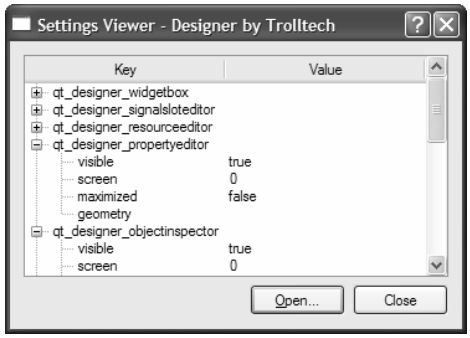


Figure 10.5
L'application Settings Viewer

Voici un extrait du constructeur :

```
SettingsViewer::SettingsViewer(QWidget *parent)
   : QDialog(parent)
{
   organization = "Trolltech";
   application = "Designer";

   treeWidget = new QTreeWidget;
   treeWidget->setColumnCount(2);
   treeWidget->setHeaderLabels(
         QStringList() << tr("Key") << tr("Value"));
   treeWidget->header()->setResizeMode(0, QHeaderView::Stretch);
   treeWidget->header()->setResizeMode(1, QHeaderView::Stretch);
   ...
   setWindowTitle(tr("Settings Viewer"));
   readSettings();
}
```

Pour accéder aux paramètres d'une application, un objet `QSettings` doit être créé avec le nom de l'organisation et le nom de l'application comme paramètres. Nous définissons des noms par défaut ("Designer" par "Trolltech"), puis nous construisons un nouveau `QTreeWidget`. Pour terminer, nous appelons la fonction `readSettings()`.

```
void SettingsViewer::readSettings()
{
   QSettings settings(organization, application);

   treeWidget->clear();
   addChildSettings(settings, 0, "");

   treeWidget->sortByColumn(0);
   treeWidget->setFocus();
   setWindowTitle(tr("Settings Viewer - %1 by %2")
                  .arg(application).arg(organization));
}
```

Les paramètres d'application sont stockés dans une hiérarchie de clés et de valeurs. La fonction privée `addChildSettings()` reçoit un objet `settings`, un parent `QTreeWidgetItem` et le "groupe" en cours. Un groupe est l'équivalent `QSettings` d'un répertoire de système de fichiers. La fonction `addChildSettings()` peut s'appeler elle-même de manière récursive pour faire défiler une arborescence arbitraire. Le premier appel de la fonction `readSettings()` transmet 0 comme élément parent pour représenter la racine.

```
void SettingsViewer::addChildSettings(QSettings &settings,
      QTreeWidgetItem *parent, const QString &group)
{
   QTreeWidgetItem *item;

   settings.beginGroup(group);

   foreach (QString key, settings.childKeys()) {
      if (parent) {
          item = new QTreeWidgetItem(parent);
      } else {
          item = new QTreeWidgetItem(treeWidget);
      }
      item->setText(0, key);
      item->setText(1, settings.value(key).toString());
   }
   foreach (QString group, settings.childGroups()) {
      if (parent) {
          item = new QTreeWidgetItem(parent);
      } else {
          item = new QTreeWidgetItem(treeWidget);
      }
      item->setText(0, group);
      addChildSettings(settings, item, group);
   }
   settings.endGroup();
}
```

La fonction `addChildSettings()` est utilisée pour créer tous les `QTreeWidgetItem`. Elle parcourt toutes les clés au niveau en cours dans la hiérarchie des paramètres et crée un `QTableWidgetItem` par clé. Si 0 est transmis en tant qu'élément `parent`, nous créons l'élément comme étant un enfant de `QTreeWidget` (il devient donc un élément de haut niveau) ; sinon, nous créons l'élément comme étant un enfant de `parent`. La première colonne correspond au nom de la clé et la seconde à la valeur correspondante.

La fonction parcourt ensuite chaque groupe du niveau en cours. Pour chacun d'eux, un nouveau `QTreeWidgetItem` est créé avec sa première colonne définie en nom du groupe. Puis, la fonction s'appelle elle-même de manière récursive avec l'élément de groupe comme parent pour alimenter le `QTreeWidget` avec les éléments enfants du groupe.

Les widgets d'affichage d'éléments présentés dans cette section nous permettent d'utiliser un style de programmation très similaire à celui utilisé dans les versions antérieures de Qt : lire tout un ensemble de données dans un widget d'affichage d'éléments, utiliser les objets des

éléments pour représenter les éléments de données, et (si les éléments sont modifiables) sauvegarder sur la source de données. Dans les sections suivantes, nous irons plus loin que cette approche simple et nous profiterons pleinement de l'architecture modèle/vue de Qt.

Utiliser des modèles prédéfinis

Qt propose plusieurs modèles prédéfinis à utiliser avec les classes d'affichage :

`QStringListModel`	Stocke une liste de chaînes
`QStandardItemModel`	Stocke des données hiérarchiques arbitraires
`QDirModel`	Encapsule le système de fichiers local
`QSqlQueryModel`	Encapsule un jeu de résultats SQL
`QSqlTableModel`	Encapsule une table SQL
`QSqlRelationalTableModel`	Encapsule une table SQL avec des clés étrangères
`QSortFilterProxyModel`	Trie et/ou filtre un autre modèle

Dans cette section, nous verrons comment employer `QStringListModel`, `QDirModel` et `QSortFilterProxyModel`. Les modèles SQL sont traités au Chapitre 13.

Commençons par une boîte de dialogue simple dont les utilisateurs peuvent se servir pour ajouter, supprimer et modifier un `QStringList`, où chaque chaîne représente un chef d'équipe. Celle-ci est présentée en Figure 10.6.

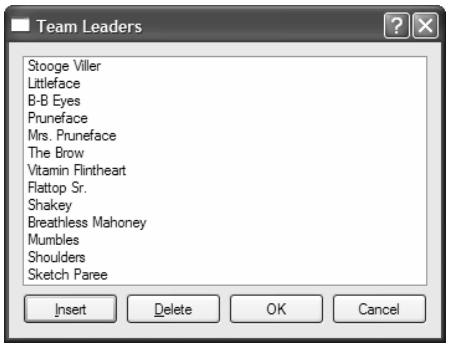


Figure 10.6
L'application Team Leaders

Voici un extrait pertinent du constructeur :

```
TeamLeadersDialog::TeamLeadersDialog(const QStringList &leaders,
                                     QWidget *parent)
```

```
    : QDialog(parent)
{
    model = new QStringListModel(this);
    model->setStringList(leaders);

    listView = new QListView;
    listView->setModel(model);
    listView->setEditTriggers(QAbstractItemView::AnyKeyPressed
                              | QAbstractItemView::DoubleClicked);
    ...
}
```

Nous créons et alimentons d'abord un `QStringListModel`. Nous créons ensuite un `QListView` et nous lui affectons comme modèle un de ceux que nous venons de créer. Nous configurons également des déclencheurs de modification pour permettre à l'utilisateur de modifier une chaîne simplement en commençant à taper quelque chose ou en double-cliquant dessus. Par défaut, aucun déclencheur de modification n'est défini sur un `QListView`, la vue est donc configurée en lecture seule.

```
void TeamLeadersDialog::insert()
{
    int row = listView->currentIndex().row();
    model->insertRows(row, 1);

    QModelIndex index = model->index(row);
    listView->setCurrentIndex(index);
    listView->edit(index);
}
```

Le slot `insert()` est invoqué lorsque l'utilisateur clique sur le bouton Insert. Le slot commence par récupérer le numéro de ligne de l'élément en cours dans la vue de liste. Chaque élément de données dans un modèle possède un "index de modèle" correspondant qui est représenté par un objet `QModelIndex`. Nous allons étudier les index de modèle plus en détail dans la prochaine section, mais pour l'instant il suffit de savoir qu'un index comporte trois composants principaux : une ligne, une colonne et un pointeur vers le modèle auquel il appartient. Pour un modèle liste unidimensionnel, la colonne est toujours 0.

Lorsque nous connaissons le numéro de ligne, nous insérons une nouvelle ligne à cet endroit. L'insertion est effectuée sur le modèle et le modèle met automatiquement à jour la vue de liste. Nous définissons ensuite l'index en cours de la vue de liste sur la ligne vide que nous venons d'insérer. Enfin, nous définissons la vue de liste en mode de modification sur la nouvelle ligne, comme si l'utilisateur avait appuyé sur une touche ou double-cliqué pour initier la modification.

```
void TeamLeadersDialog::del()
{
    model->removeRows(listView->currentIndex().row(), 1);
}
```

Dans le constructeur, le signal `clicked()` du bouton Delete est relié au slot `del()`. Vu que nous avons supprimé la ligne en cours, nous pouvons appeler `removeRows()` avec la position

actuelle d'index et un nombre de lignes de 1. Comme avec l'insertion, nous nous basons sur le modèle pour mettre à jour la vue de façon appropriée.

```
QStringList TeamLeadersDialog::leaders() const
{
   return model->stringList();
}
```

Enfin, la fonction `leaders()` procure un moyen de lire les chaînes modifiées quand la boîte de dialogue est fermée.

`TeamLeadersDialog` pourrait devenir une boîte de dialogue générique de modification de liste de chaînes simplement en paramétrant le titre de sa fenêtre. Une autre boîte de dialogue générique souvent demandée est une boîte qui présente une liste de fichiers ou de répertoires à l'utilisateur. Le prochain exemple exploite la classe `QDirModel`, qui encapsule le système de fichiers de l'ordinateur et qui peut afficher (et masquer) les divers attributs de fichiers. Ce modèle peut appliquer un filtre pour limiter les types d'entrées du système de fichiers qui sont affichées et peut organiser les entrées de plusieurs manières différentes.

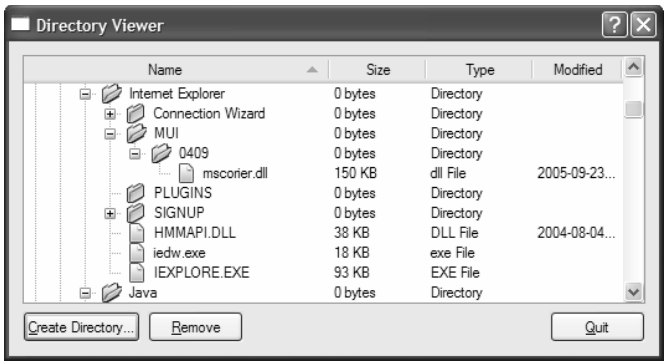


Figure 10.7
L'application Directory Viewer

Nous analyserons d'abord la création et nous configurerons le modèle et la vue dans le constructeur de la boîte de dialogue Directory Viewer (voir Figure 10.7).

```
DirectoryViewer::DirectoryViewer(QWidget *parent)
   : QDialog(parent)
{
   model = new QDirModel;
   model->setReadOnly(false);
   model->setSorting(QDir::DirsFirst | QDir::IgnoreCase | QDir::Name);

   treeView = new QTreeView;
   treeView->setModel(model);
   treeView->header()->setStretchLastSection(true);
   treeView->header()->setSortIndicator(0, Qt::AscendingOrder);
   treeView->header()->setSortIndicatorShown(true);
   treeView->header()->setClickable(true);
```

```
    QModelIndex index = model->index(QDir::currentPath());
    treeView->expand(index);
    treeView->scrollTo(index);
    treeView->resizeColumnToContents(0);
    ...
}
```

Lorsque le modèle a été construit, nous faisons le nécessaire pour qu'il puisse être modifié et nous définissons les divers attributs d'ordre de tri. Nous créons ensuite le `QTreeView` qui affichera les données du modèle. L'en-tête du `QTreeView` peut être utilisé pour proposer un tri contrôlé par l'utilisateur. Si cet en-tête est cliquable, l'utilisateur est en mesure de trier n'importe quelle colonne en cliquant sur ce dernier ; en cliquant plusieurs fois dessus, il choisit entre les tris croissants et décroissants. Une fois que l'en-tête de l'arborescence a été configuré, nous obtenons l'index de modèle du répertoire en cours et nous sommes sûrs que ce répertoire est visible en développant ses parents si nécessaire à l'aide de `expand()` et en le localisant grâce à `scrollTo()`. Nous nous assurons également que la première colonne est suffisamment grande pour afficher toutes les entrées sans utiliser de points de suspension (...).

Dans la partie du code du constructeur qui n'est pas présentée ici, nous avons connecté les boutons Create Directory (Créer un répertoire) et Remove (Supprimer) aux slots pour effectuer ces actions. Nous n'avons pas besoin de bouton Rename parce que les utilisateurs peuvent renommer directement en appuyant sur F2 et en tapant du texte.

```
void DirectoryViewer::createDirectory()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid())
        return;

    QString dirName = QInputDialog::getText(this,
                                tr("Create Directory"),
                                tr("Directory name"));

    if (!dirName.isEmpty()) {
        if (!model->mkdir(index, dirName).isValid())
            QMessageBox::information(this, tr("Create Directory"),
                     tr("Failed to create the directory"));
    }
}
```

Si l'utilisateur entre un nom de répertoire dans la boîte de dialogue, nous essayons de créer un répertoire avec ce nom comme enfant du répertoire en cours. La fonction `QDirModel::mkdir()` reçoit l'index du répertoire parent et le nom du nouveau répertoire, et retourne l'index de modèle du répertoire qu'il a créé. Si l'opération échoue, elle retourne un index de modèle invalide.

```
void DirectoryViewer::remove()
{
    QModelIndex index = treeView->currentIndex();
```

```
    if (!index.isValid())
        return;

    bool ok;
    if (model->fileInfo(index).isDir()) {
        ok = model->rmdir(index);
    } else {
        ok = model->remove(index);
    }
    if (!ok)
        QMessageBox::information(this, tr("Remove"),
                 tr("Failed to remove %1").arg(model->fileName(index)));
}
```

Si l'utilisateur clique sur Remove, nous tentons de supprimer le fichier ou le répertoire associé à l'élément en cours. Pour ce faire, nous pourrions utiliser `QDir`, mais `QDirModel` propose des fonctions pratiques qui fonctionnent avec `QModelIndexes`.

Le dernier exemple de cette section vous montre comment employer `QSortFilterProxyModel`. Contrairement aux autres modèles prédéfinis, ce modèle encapsule un modèle existant et manipule les données qui sont transmises entre le modèle sous-jacent et la vue. Dans notre exemple, le modèle sous-jacent est un `QStringListModel` initialisé avec la liste des noms de couleur reconnues par Qt (obtenue *via* `QColor::colorNames()`). L'utilisateur peut saisir une chaîne de filtre dans un `QLineEdit` et spécifier la manière dont cette chaîne doit être interprétée (comme une expression régulière, un modèle générique ou une chaîne fixe) grâce à une zone de liste déroulante (voir Figure 10.8).

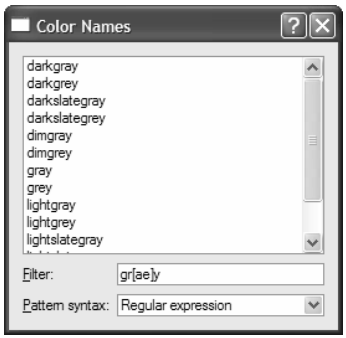


Figure 10.8
L'application Color Names

Voici un extrait du constructeur de `ColorNamesDialog` :

```
ColorNamesDialog::ColorNamesDialog(QWidget *parent)
    : QDialog(parent)
{
    sourceModel = new QStringListModel(this);
    sourceModel->setStringList(QColor::colorNames());

    proxyModel = new QSortFilterProxyModel(this);
```

```
    proxyModel->setSourceModel(sourceModel);
    proxyModel->setFilterKeyColumn(0);

    listView = new QListView;
    listView->setModel(proxyModel);
    ...
    syntaxComboBox = new QComboBox;
    syntaxComboBox->addItem(tr("Regular expression"), QRegExp::RegExp);
    syntaxComboBox->addItem(tr("Wildcard"), QRegExp::Wildcard);
    syntaxComboBox->addItem(tr("Fixed string"), QRegExp::FixedString);
    ...
}
```

`QStringListModel` est créé et alimenté de manière habituelle. Puis, nous construisons `QSortFilterProxyModel`. Nous transmettons le modèle sous-jacent à l'aide de `setSourceModel()` et nous demandons au proxy de filtrer en se basant sur la colonne 0 du modèle original. La fonction `QComboBox::addItem()` reçoit un argument facultatif "donnée" de type `QVariant` ; nous l'utilisons pour enregistrer la valeur `QRegExp::PatternSyntax` qui correspond au texte de chaque élément.

```
void ColorNamesDialog::reapplyFilter()
{
    QRegExp::PatternSyntax syntax =
            QRegExp::PatternSyntax(syntaxComboBox->itemData(
                    syntaxComboBox->currentIndex()).toInt());
    QRegExp regExp(filterLineEdit->text(), Qt::CaseInsensitive, syntax);
    proxyModel->setFilterRegExp(regExp);
}
```

Le slot `reapplyFilter()` est invoqué dès que l'utilisateur modifie la chaîne de filtre ou la zone de liste déroulante correspondant au modèle. Nous créons un `QRegExp` en utilisant le texte présent dans l'éditeur de lignes. Nous faisons ensuite correspondre la syntaxe de son modèle à celle stockée dans les données de l'élément en cours dans la zone de liste déroulante relative à la syntaxe. Puis nous appelons `setFilterRegExp()`, le nouveau filtre s'active et la vue est mise à jour automatiquement.

Implémenter des modèles personnalisés

Les modèles prédéfinis de Qt sont pratiques pour gérer et afficher des données. Cependant, certaines sources de données ne peuvent pas être utilisées efficacement avec les modèles prédéfinis, c'est pourquoi il est parfois nécessaire de créer des modèles personnalisés optimisés pour la source de données sous-jacente.

Avant de commencer à créer des modèles personnalisés, analysons d'abord les concepts essentiels utilisés dans l'architecture modèle/vue de Qt. Chaque élément de données dans un modèle possède un index de modèle et un ensemble d'attributs, appelés rôles, qui peuvent prendre des valeurs arbitraires. Nous avons vu précédemment que les rôles les plus couramment employés

sont `Qt::EditRole` et `Qt::DisplayRole`. D'autres rôles sont utilisés pour des données supplémentaires (par exemple `Qt::ToolTipRole`, `Qt::StatusTipRole` et `Qt::WhatsThisRole`) et d'autres encore pour contrôler les attributs d'affichage de base (tels que `Qt::FontRole`, `Qt::TextAlignmentRole`, `Qt::TextColorRole` et `Qt::BackgroundColorRole`).

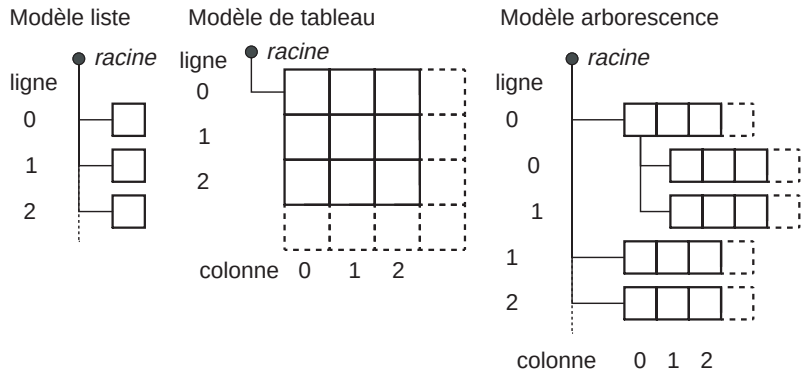


Figure 10.9
Vue schématique des modèles de Qt

Pour un modèle liste, le seul composant d'index pertinent est le nombre de lignes, accessible depuis `QModelIndex::row()`. Pour un modèle de tableau, les composants d'index pertinents sont les nombres de lignes et de colonnes, accessibles depuis `QModelIndex::row()` et `QModelIndex::column()`. Pour les modèles liste et tableau, le parent de chaque élément est la racine, qui est représentée par un `QModelIndex` invalide. Les deux premiers exemples de cette section vous montrent comment implémenter des modèles de tableau personnalisés.

Un modèle arborescence ressemble à un modèle de tableau, à quelques différences près. Comme un modèle de tableau, la racine est le parent des éléments de haut niveau (un `QModelIndex` invalide), mais le parent de tout autre élément est un autre élément dans la hiérarchie. Les parents sont accessibles depuis `QModelIndex::parent()`. Chaque élément possède ses données de rôle et aucun ou plusieurs enfants, chacun étant un élément en soi. Vu que les éléments peuvent avoir d'autres éléments comme enfants, il est possible de représenter des structures de données récursives (à la façon d'une arborescence), comme vous le montrera le dernier exemple de cette section.

Le premier exemple de cette section est un modèle de tableau en lecture seule qui affiche des valeurs monétaires en relation les unes avec les autres (voir Figure 10.10).

L'application pourrait être implémentée à partir d'un simple tableau, mais nous voulons nous servir d'un modèle personnalisé pour profiter de certaines propriétés des données qui minimisent le stockage. Si nous voulions conserver les 162 devises actuellement cotées dans un tableau, nous devrions stocker $162 \times 162 = 26\,244$ valeurs ; avec le modèle personnalisé présenté ci-après, nous n'enregistrons que 162 valeurs (la valeur de chaque devise par rapport au dollar américain).

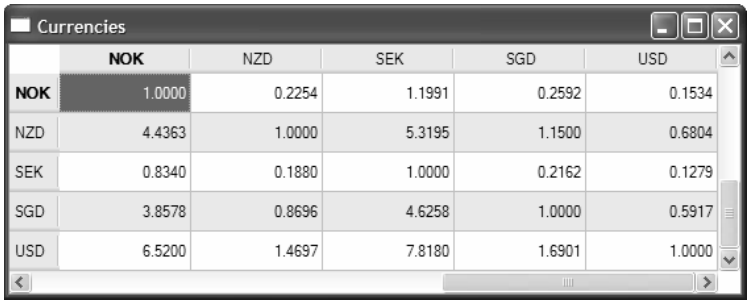


Figure 10.10
L'application Currencies

La classe `CurrencyModel` sera utilisée avec un `QTableView` standard. Elle est alimentée avec un `QMap<QString,double>` ; chaque clé correspond au code de la devise et chaque valeur correspond à la valeur de la devise en dollars américains. Voici un extrait de code qui montre comment le tableau de correspondance est alimenté et comment le modèle est utilisé :

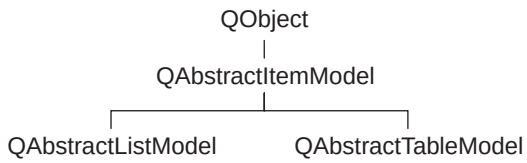
```
QMap<QString, double> currencyMap;
currencyMap.insert("AUD", 1.3259);
currencyMap.insert("CHF", 1.2970);
...
currencyMap.insert("SGD", 1.6901);
currencyMap.insert("USD", 1.0000);

CurrencyModel currencyModel;
currencyModel.setCurrencyMap(currencyMap);

QTableView tableView;
tableView.setModel(&currencyModel);
tableView.setAlternatingRowColors(true);
```

Etudions désormais l'implémentation du modèle, en commençant par son en-tête :

```
class CurrencyModel : public QAbstractTableModel
{
public:
   CurrencyModel(QObject *parent = 0);

   void setCurrencyMap(const QMap<QString, double> &map);
   int rowCount(const QModelIndex &parent) const;
   int columnCount(const QModelIndex &parent) const;
   QVariant data(const QModelIndex &index, int role) const;
   QVariant headerData(int section, Qt::Orientation orientation,
                       int role) const;

private:
   QString currencyAt(int offset) const;

   QMap<QString, double> currencyMap;
};
```

Nous avons choisi de dériver `QAbstractTableModel` pour notre modèle, parce que cela correspond le plus à notre source de données. Qt propose plusieurs classes de base de modèle, y compris `QAbstractListModel`, `QAbstractTableModel` et `QAbstractItemModel` (voir Figure 10.11). La classe `QAbstractItemModel` est employée pour supporter une grande variété de modèles, dont ceux qui se basent sur des structures de données récursives, alors que les classes `QAbstractListModel` et `QAbstractTableModel` sont proposées pour une question de commodité lors de l'utilisation d'ensembles de données à une ou deux dimensions.

Figure 10.11
Arbre d'héritage des classes de modèle abstraites



Pour un modèle de tableau en lecture seule, nous devons réimplémenter trois fonctions : `rowCount()`, `columnCount()` et `data()`. Dans ce cas, nous avons aussi réimplémenté `headerData()` et nous fournissons une fonction pour initialiser les données (`setCurrencyMap()`).

```
CurrencyModel::CurrencyModel(QObject *parent)
   : QAbstractTableModel(parent)
{
}
```

Nous n'avons pas besoin de faire quoi que ce soit dans le constructeur, sauf transmettre le paramètre `parent` à la classe de base.

```
int CurrencyModel::rowCount(const QModelIndex & /* parent */) const
{
   return currencyMap.count();
}

int CurrencyModel::columnCount(const QModelIndex & /* parent */) const
{
   return currencyMap.count();
}
```

Pour ce modèle de tableau, les nombres de lignes et de colonnes correspondent aux nombres de devises dans le tableau de correspondance des devises. Le paramètre `parent` n'a aucune signification pour un modèle de tableau ; il est présent parce que `rowCount()` et `columnCount()` sont hérités de la classe de base `QAbstractItemModel` plus générique, qui prend en charge les hiérarchies.

```
QVariant CurrencyModel::data(const QModelIndex &index, int role) const
{
   if (!index.isValid())
      return QVariant();
```

```
    if (role == Qt::TextAlignmentRole) {
        return int(Qt::AlignRight | Qt::AlignVCenter);
    } else if (role == Qt::DisplayRole) {
        QString rowCurrency = currencyAt(index.row());
        QString columnCurrency = currencyAt(index.column());

        if (currencyMap.value(rowCurrency) == 0.0)
            return "####";

        double amount = currencyMap.value(columnCurrency)
                        / currencyMap.value(rowCurrency);

        return QString("%1").arg(amount, 0, 'f', 4);
    }
    return QVariant();
}
```

La fonction `data()` retourne la valeur de n'importe quel rôle d'un élément. L'élément est spécifié sous forme de `QModelIndex`. Pour un modèle de tableau, les composants intéressants d'un `QModelIndex` sont ses nombres de lignes et de colonnes, disponibles grâce à `row()` et `column()`.

Si le rôle est `Qt::TextAlignmentRole`, nous retournons un alignement adapté aux nombres. Si le rôle d'affichage est `Qt::DisplayRole`, nous recherchons la valeur de chaque devise et nous calculons le taux de change.

Nous pourrions retourner la valeur calculée sous forme de type `double`, mais nous n'aurions aucun contrôle sur le nombre de chiffres après la virgule (à moins d'utiliser un délégué personnalisé). Nous retournons donc plutôt la valeur sous forme de chaîne, mise en forme comme nous le souhaitons.

```
QVariant CurrencyModel::headerData(int section,
                                   Qt::Orientation /* orientation */,
                                   int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();
    return currencyAt(section);
}
```

La fonction `headerData()` est appelée par la vue pour alimenter ses en-têtes verticaux et horizontaux. Le paramètre `section` correspond au nombre de lignes ou de colonnes (selon l'orientation). Vu que les lignes et les colonnes ont les mêmes codes de devise, nous ne nous soucions pas de l'orientation et nous retournons simplement le code de la devise pour le numéro de section donné.

```
void CurrencyModel::setCurrencyMap(const QMap<QString, double> &map)
{
    currencyMap = map;
    reset();
}
```

L'appelant peut modifier le tableau de correspondance des devises en exécutant `setCurrencyMap()`. L'appel de `QAbstractItemModel::reset()` informe n'importe quelle vue qui utilise le modèle que toutes leurs données sont invalides ; ceci les oblige à demander des données actualisées pour les éléments visibles.

```
QString CurrencyModel::currencyAt(int offset) const
{
   return (currencyMap.begin() + offset).key();
}
```

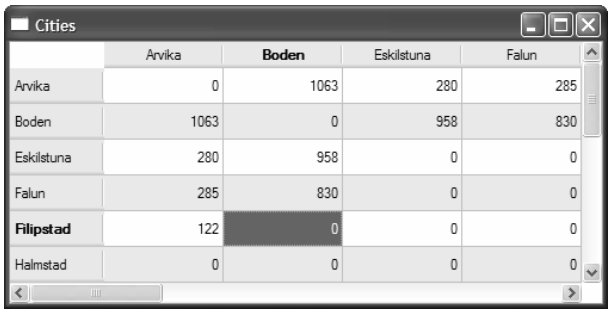
La fonction `currencyAt()` retourne la clé (le code de la devise) à la position donnée dans le tableau de correspondance des devises. Nous utilisons un itérateur de style STL pour trouver l'élément et appeler `key()`.

Comme nous venons de le voir, il n'est pas difficile de créer des modèles en lecture seule, et en fonction de la nature des données sous-jacentes, il est possible d'économiser de la mémoire et d'accélérer les temps de réponse avec un modèle bien conçu. Le prochain exemple, l'application Cities, se base aussi sur un tableau, mais cette fois-ci les données sont saisies par l'utilisateur (voir Figure 10.12).

Cette application est utilisée pour enregistrer des valeurs indiquant la distance entre deux villes. Comme l'exemple précédent, nous pourrions simplement utiliser un `QTableWidget` et stocker un élément pour chaque paire de villes. Cependant, un modèle personnalisé pourrait être plus efficace, parce que la distance entre une ville A et une ville B est la même que vous alliez de A à B ou de B à A, les éléments se reflètent donc le long de la diagonale principale.

Pour voir comment un modèle personnalisé se compare à un simple tableau, supposons que nous avons trois villes, A, B et C. Si nous conservions une valeur pour chaque combinaison, nous devrions stocker neuf valeurs. Un modèle bien conçu ne nécessiterait que trois éléments (A, B), (A, C) et (B, C).

Figure 10.12
L'application Cities



Voici comment nous avons configuré et exploité le modèle :

```
QStringList cities;
cities << "Arvika" << "Boden" << "Eskilstuna" << "Falun"
       << "Filipstad" << "Halmstad" << "Helsingborg" << "Karlstad"
```

```
            << "Kiruna" << "Kramfors" << "Motala" << "Sandviken"
            << "Skara" << "Stockholm" << "Sundsvall" << "Trelleborg";

    CityModel cityModel;
    cityModel.setCities(cities);

    QTableView tableView;
    tableView.setModel(&cityModel);
    tableView.setAlternatingRowColors(true);
```

Nous devons réimplémenter les mêmes fonctions que pour l'exemple précédent. De plus, nous devons aussi réimplémenter `setData()` et `flags()` pour que le modèle puisse être modifié. Voici la définition de classe :

```
class CityModel : public QAbstractTableModel
{
    Q_OBJECT

public:
    CityModel(QObject *parent = 0);

    void setCities(const QStringList &cityNames);
    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    bool setData(const QModelIndex &index, const QVariant &value,
                 int role);
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role) const;
    Qt::ItemFlags flags(const QModelIndex &index) const;

private:
    int offsetOf(int row, int column) const;

    QStringList cities;
    QVector<int> distances;
};
```

Pour ce modèle, nous utilisons deux structures de données : `cities` de type `QStringList` pour contenir les noms de ville, et `distances` de type `QVector<int>` pour enregistrer la distance entre chaque paire unique de villes.

```
CityModel::CityModel(QObject *parent)
    : QAbstractTableModel(parent)
{
}
```

Le constructeur ne fait rien à part transmettre le paramètre `parent` à la classe de base.

```
int CityModel::rowCount(const QModelIndex & /* parent */) const
{
    return cities.count();
```

```
}

int CityModel::columnCount(const QModelIndex & /* parent */) const
{
   return cities.count();
}
```

Vu que nous avons une grille carrée de villes, le nombre de lignes et de colonnes correspond au nombre de villes de notre liste.

```
QVariant CityModel::data(const QModelIndex &index, int role) const
{
   if (!index.isValid())
      return QVariant();

   if (role == Qt::TextAlignmentRole) {
      return int(Qt::AlignRight | Qt::AlignVCenter);
   } else if (role == Qt::DisplayRole) {
      if (index.row() == index.column())
         return 0;
      int offset = offsetOf(index.row(), index.column());
      return distances[offset];
   }
   return QVariant();
}
```

La fonction `data()` est similaire à ce que nous effectué dans `CurrencyModel`. Elle retourne 0 si la ligne et la colonne sont identiques, parce que cela correspond au cas où les deux villes sont les mêmes ; sinon elle recherche l'entrée de la ligne et de la colonne dans le vecteur `distances` et renvoie la distance pour cette paire de villes particulière.

```
QVariant CityModel::headerData(int section,
                               Qt::Orientation /* orientation */,
                               int role) const
{
   if (role == Qt::DisplayRole)
      return cities[section];
   return QVariant();
}
```

La fonction `headerData()` est simple puisque nous avons un tableau carré où chaque ligne possède un en-tête de colonne identique. Nous retournons simplement le nom de la ville à la position donnée dans la liste de chaîne `cities`.

```
bool CityModel::setData(const QModelIndex &index,
                        const QVariant &value, int role)
{
   if (index.isValid() && index.row() != index.column()
         && role == Qt::EditRole) {
      int offset = offsetOf(index.row(), index.column());
      distances[offset] = value.toInt();
```

```
        QModelIndex transposedIndex = createIndex(index.column(),
                                                  index.row());
        emit dataChanged(index, index);
        emit dataChanged(transposedIndex, transposedIndex);
        return true;
    }
    return false;
}
```

La fonction `setData()` est invoquée quand l'utilisateur modifie un élément. En supposant que l'index de modèle est valide, que les deux villes sont différentes et que l'élément de données à modifier est `Qt::EditRole`, la fonction stocke la valeur que l'utilisateur a saisie dans le vecteur `distances`.

La fonction `createIndex()` sert à générer un index de modèle. Nous en avons besoin pour obtenir l'index de modèle de l'élément symétrique de l'élément configuré par rapport à la diagonale principale, vu que les deux éléments doivent afficher les mêmes données. La fonction `createIndex()` reçoit la ligne avant la colonne ; ici, nous inversons les paramètres pour obtenir l'index de modèle de l'élément symétrique à celui spécifié par `index`.

Nous émettons le signal `dataChanged()` avec l'index de modèle de l'élément qui a été modifié. Ce signal reçoit deux index de modèle, parce qu'un changement peut affecter une région rectangulaire constituée de plusieurs lignes et colonnes. Les index transmis représentent l'élément situé en haut à gauche et l'élément en bas à droite de la zone modifiée. Nous émettons aussi le signal `dataChanged()` à l'attention de l'index transposé afin que la vue actualise l'affichage de l'élément. Enfin, nous retournons `true` ou `false` pour indiquer si la modification a été effectuée avec succès ou non.

```
Qt::ItemFlags CityModel::flags(const QModelIndex &index) const
{
    Qt::ItemFlags flags = QAbstractItemModel::flags(index);
    if (index.row() != index.column())
        flags |= Qt::ItemIsEditable;
    return flags;
}
```

Le modèle se sert de la fonction `flags()` pour annoncer les possibilités d'action sur l'élément (par exemple, s'il peut être modifié ou non). L'implémentation par défaut de `QAbstractTableModel` retourne `Qt::ItemIsSelectable | Qt::ItemIsEnabled`. Nous ajoutons l'indicateur `Qt::ItemIsEditable` pour tous les éléments sauf ceux qui se trouvent sur les diagonales (qui sont toujours nuls).

```
void CityModel::setCities(const QStringList &cityNames)
{
    cities = cityNames;
    distances.resize(cities.count() * (cities.count() - 1) / 2);
    distances.fill(0);
    reset();
}
```

Si nous recevons une nouvelle liste de villes, nous définissons le `QStringList` privé en nouvelle liste, nous redimensionnons et nous effaçons le vecteur `distances` puis nous appelons `QAbstractItemModel::reset()` pour informer toutes les vues que leurs éléments visibles doivent être à nouveau récupérés.

```
int CityModel::offsetOf(int row, int column) const
{
   if (row < column)
      qSwap(row, column);
   return (row * (row - 1) / 2) + column;
}
```

La fonction privée `offsetOf()` calcule l'index d'une paire de villes donnée dans le vecteur `distances`. Par exemple, si nous avions les villes A, B, C et D et si l'utilisateur avait mis à jour la ligne 3, colonne 1, B à D, le décalage serait de 3 _ (3 – 1)/2 + 1 = 4. Si l'utilisateur avait mis à jour la ligne 1, colonne 3, D à B, grâce à `qSwap()`, exactement le même calcul aurait été accompli et un décalage identique aurait été retourné.

Figure 10.13
Les structures de données `cities` et `distances` et le modèle de tableau

Villes

A	B	C	D

Distances

A÷ B	A÷ C	A÷ D	B÷ C	B÷ D	C÷ D

Modèle de tableau

	A	B	C	D
A	0	A÷ B	A÷ C	A÷ D
B	A÷ B	0	B÷ C	B÷ D
C	A÷ C	B÷ C	0	C÷ D
D	A÷ D	B÷ D	C÷ D	0

Le dernier exemple de cette section est un modèle qui présente l'arbre d'analyse d'une expression régulière donnée. Une expression régulière est constituée d'un ou plusieurs termes, séparés par des caractères "|". L'expression régulière "alpha|bravo|charlie" contient donc trois termes. Chaque terme est une séquence d'un ou plusieurs facteurs ; par exemple, le terme "bravo" est composé de cinq facteurs (chaque lettre est un facteur). Les facteurs peuvent encore être décomposés en atome et en quantificateur facultatif, comme "*", "+" et "?".Vu que les expressions régulières peuvent contenir des sous-expressions entre parenthèses, les arbres d'analyse correspondants seront récursifs.

L'expression régulière présentée en Figure 10.14, "ab|(cd)?e", correspond à un 'a' suivi d'un 'b', ou d'un 'c' suivi d'un 'd' puis d'un 'e', ou simplement d'un 'e'. Elle correspondra ainsi à "ab" et "cde", mais pas à "bc" ou "cd".

L'application Regexp Parser se compose de quatre classes :

- `RegExpWindow` est une fenêtre qui permet à l'utilisateur de saisir une expression régulière et qui affiche l'arbre d'analyse correspondant.
- `RegExpParser` génère un arbre d'analyse à partir d'une expression régulière.
- `RegExpModel` est un modèle d'arborescence qui encapsule un arbre d'analyse.
- `Node` représente un élément dans un arbre d'analyse.

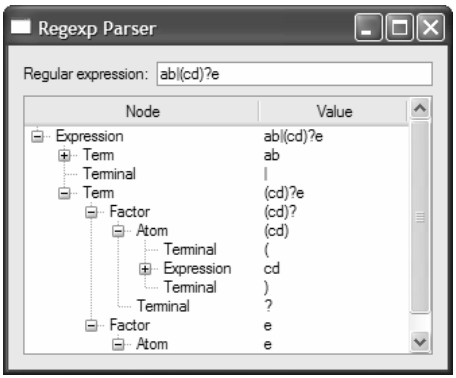


Figure 10.14
L'application Regexp Parser

Commençons par la classe `Node` :

```
class Node
{
public:
   enum Type { RegExp, Expression, Term, Factor, Atom, Terminal };

   Node(Type type, const QString &str = "");
   ~Node();

   Type type;
   QString str;
   Node *parent;
   QList<Node *> children;
};
```

Chaque nœud possède un type, une chaîne (qui peut être vide), un parent (qui peut être 0) et une liste de nœuds enfants (qui peut être vide).

```
Node::Node(Type type, const QString &str)
{
   this->type = type;
   this->str = str;
   parent = 0;
}
```

Le constructeur initialise simplement le type et la chaîne du nœud. Etant donné que toutes les données sont publiques, le code qui utilise `Node` peut opérer directement sur le type, la chaîne, le parent et les enfants.

```
Node::~Node()
{
   qDeleteAll(children);
}
```

La fonction `qDeleteAll()` parcourt un conteneur de pointeurs et appelle `delete` sur chacun d'eux. Elle ne définit pas les pointeurs en 0, donc si elle est utilisée en dehors d'un destructeur, il est fréquent de la voir suivie d'un appel de `clear()` sur le conteneur qui renferme les pointeurs.

Maintenant que nous avons défini nos éléments de données (chacun représenté par un `Node`), nous sommes prêts à créer un modèle :

```
class RegExpModel : public QAbstractItemModel
{
public:
   RegExpModel(QObject *parent = 0);
   ~RegExpModel();

   void setRootNode(Node *node);

   QModelIndex index(int row, int column,
                     const QModelIndex &parent) const;
   QModelIndex parent(const QModelIndex &child) const;

   int rowCount(const QModelIndex &parent) const;
   int columnCount(const QModelIndex &parent) const;
   QVariant data(const QModelIndex &index, int role) const;
   QVariant headerData(int section, Qt::Orientation orientation,
                       int role) const;

private:
   Node *nodeFromIndex(const QModelIndex &index) const;

   Node *rootNode;
};
```

Cette fois-ci nous avons hérité de `QAbstractItemModel` plutôt que de sa sous-classe dédiée `QAbstractTableModel`, parce que nous voulons créer un modèle hiérarchique. Les fonctions essentielles que nous devons réimplémenter sont toujours les mêmes, sauf que nous devons aussi implémenter `index()` et `parent()`. Pour définir les données du modèle, une fonction `setRootNode()` doit être invoquée avec le nœud racine de l'arbre d'analyse.

```
RegExpModel::RegExpModel(QObject *parent)
   : QAbstractItemModel(parent)
{
   rootNode = 0;
}
```

Dans le constructeur du modèle, nous n'avons qu'à configurer le nœud racine en valeur nulle et transmettre le `parent` à la classe de base.

```
RegExpModel::~RegExpModel()
{
   delete rootNode;
}
```

Dans le destructeur, nous supprimons le nœud racine. Si le nœud racine a des enfants, chacun d'eux est supprimé par le destructeur `Node`, et ainsi de suite de manière récursive.

```
void RegExpModel::setRootNode(Node *node)
{
   delete rootNode;
   rootNode = node;
   reset();
}
```

Quand un nouveau nœud racine est défini, nous supprimons d'abord tout nœud racine précédent (et tous ses enfants). Nous configurons ensuite le nouveau nœud racine et nous appelons `reset()` pour informer les vues qu'elles doivent à nouveau récupérer les données des éléments visibles.

```
QModelIndex RegExpModel::index(int row, int column,
                               const QModelIndex &parent) const
{
   if (!rootNode)
      return QModelIndex();
   Node *parentNode = nodeFromIndex(parent)
   return createIndex(row, column, parentNode->children[row]);
}
```

La fonction `index()` est réimplémentée dans `QAbstractItemModel`. Elle est appelée dès que le modèle ou la vue doit créer un `QModelIndex` pour un élément enfant particulier (ou un élément de haut niveau si `parent` est un `QModelIndex` invalide). Pour les modèles de tableau et de liste, nous n'avons pas besoin de réimplémenter cette fonction, parce que les implémentations par défaut de `QAbstractListModel` et `QAbstractTableModel` sont normalement suffisantes.

Dans notre implémentation d'`index()`, si aucun arbre d'analyse n'est configuré, nous retournons un `QModelIndex` invalide. Sinon, nous créons un `QModelIndex` avec la ligne et la colonne données et avec un `Node *` pour l'enfant demandé. S'agissant des modèles hiérarchiques, il n'est pas suffisant de connaître la ligne et la colonne d'un élément par rapport à son parent pour l'identifier ; nous devons aussi savoir *qui* est son parent. Pour résoudre ce problème, nous pouvons stocker un pointeur vers le nœud interne dans le `QModelIndex`. `QModelIndex` nous permet de conserver un `void *` ou un `int` en plus des nombres de lignes et de colonnes.

Le `Node *` de l'enfant est obtenu par le biais de la liste `children` du nœud parent. Le nœud parent est extrait de l'index de modèle `parent` grâce à la fonction privée `nodeFromIndex()` :

```
Node *RegExpModel::nodeFromIndex(const QModelIndex &index) const
{
   if (index.isValid()) {
       return static_cast<Node *>(index.internalPointer());
   } else {
       return rootNode;
   }
}
```

La fonction `nodeFromIndex()` convertit le `void *` de l'index donné en `Node *` ou retourne le nœud racine si l'index est invalide, puisqu'un index de modèle invalide représente la racine dans un modèle.

```
int RegExpModel::rowCount(const QModelIndex &parent) const
{
   Node *parentNode = nodeFromIndex(parent);
   if (!parentNode)
      return 0;
   return parentNode->children.count();
}
```

Le nombre de lignes d'un élément particulier correspond simplement au nombre d'enfants qu'il possède.

```
int RegExpModel::columnCount(const QModelIndex & /* parent */) const
{
   return 2;
}
```

Le nombre de colonnes est fixé à 2. La première colonne contient les types de nœuds ; la seconde comporte les valeurs des nœuds.

```
QModelIndex RegExpModel::parent(const QModelIndex &child) const
{
   Node *node = nodeFromIndex(child);
   if (!node)
      return QModelIndex();
   Node *parentNode = node->parent;
   if (!parentNode)
      return QModelIndex();
   Node *grandparentNode = parentNode->parent;
   if (!grandparentNode)
      return QModelIndex();

   int row = grandparentNode->children.indexOf(parentNode);
   return createIndex(row, child.column(), parentNode);
}
```

Récupérer le parent `QModelIndex` d'un enfant est un peu plus complexe que de rechercher l'enfant d'un parent. Nous pouvons facilement récupérer le nœud parent à l'aide de `nodeFromIndex()` et poursuivre en utilisant le pointeur du parent de `Node`, mais pour obtenir le numéro de ligne (la position du parent parmi ses pairs), nous devons remonter jusqu'au grand-parent et rechercher la position d'index du parent dans la liste des enfants de son parent (c'est-à-dire celle du grand-parent de l'enfant).

```
QVariant RegExpModel::data(const QModelIndex &index, int role) const
{
   if (role != Qt::DisplayRole)
      return QVariant();
```

```
    Node *node = nodeFromIndex(index);
    if (!node)
        return QVariant();

    if (index.column() == 0) {
        switch (node->type) {
        case Node::RegExp:
            return tr("RegExp");
        case Node::Expression:
            return tr("Expression");
        case Node::Term:
            return tr("Term");
        case Node::Factor:
            return tr("Factor");
        case Node::Atom:
            return tr("Atom");
        case Node::Terminal:
            return tr("Terminal");
        default:
            return tr("Unknown");
        }
    } else if (index.column() == 1) {
        return node->str;
    }
    return QVariant();
}
```

Dans `data()`, nous récupérons le `Node *` de l'élément demandé et nous nous en servons pour accéder aux données sous-jacentes. Si l'appelant veut une valeur pour n'importe quel rôle excepté `Qt::DisplayRole` ou s'il ne peut pas récupérer un `Node` pour l'index de modèle donné, nous retournons un `QVariant` invalide. Si la colonne est 0, nous renvoyons le nom du type du nœud ; si la colonne est 1, nous retournons la valeur du nœud (sa chaîne).

```
QVariant RegExpModel::headerData(int section,
                                 Qt::Orientation orientation,
                                 int role) const
{
    if (orientation == Qt::Horizontal && role == Qt::DisplayRole) {
        if (section == 0) {
            return tr("Node");
        } else if (section == 1) {
            return tr("Value");
        }
    }
    return QVariant();
}
```

Dans notre réimplémentation de `headerData()`, nous retournons les intitulés appropriés des en-têtes horizontaux. La classe `QTreeView`, qui est employée pour visualiser des modèles hiérarchiques, ne possède pas d'en-tête vertical, nous ignorons donc cette éventualité.

Maintenant que nous avons étudié les classes `Node` et `RegExpModel`, voyons comment le nœud racine est créé quand l'utilisateur modifie le texte dans l'éditeur de lignes :

```
void RegExpWindow::regExpChanged(const QString &regExp)
{
   RegExpParser parser;
   Node *rootNode = parser.parse(regExp);
   regExpModel->setRootNode(rootNode);
}
```

Quand l'utilisateur change le texte dans l'éditeur de lignes de l'application, le slot `regExpChanged()` de la fenêtre principale est appelé. Dans ce slot, le texte de l'utilisateur est analysé et l'analyseur retourne un pointeur vers le nœud racine de l'arbre d'analyse.

Nous n'avons pas étudié la classe `RegExpParser` parce qu'elle n'est pas pertinente pour les interfaces graphiques ou la programmation modèle/vue. Le code source complet de cet exemple se trouve sur la page dédiée à cet ouvrage sur le site web de Pearson, www.pearson.fr.

Dans cette section, nous avons vu comment créer trois modèles personnalisés différents. De nombreux modèles sont beaucoup plus simples que ceux présentés ici, avec des correspondances uniques entre les éléments et les index de modèle. D'autres exemples modèle/vue sont fournis avec Qt, accompagnés d'une documentation détaillée.

Implémenter des délégués personnalisés

Les éléments individuels dans les vues sont affichés et modifiés à l'aide de délégués. Dans la majorité des cas, le délégué par défaut proposé par une vue s'avère suffisant. Si nous voulons contrôler davantage l'affichage des éléments, nous pouvons atteindre notre objectif simplement en utilisant un modèle personnalisé : dans notre réimplémentation de `data()`, nous avons la possibilité de gérer `Qt::FontRole`, `Qt::TextAlignmentRole`, `Qt::TextColorRole` et `Qt::BackgroundColorRole` et ceux-ci sont employés par le délégué par défaut. Par exemple, dans les exemples Cities et Currencies présentés auparavant, nous avons géré `Qt::TextAlignmentRole` pour obtenir des nombres justifiés à droite.

Si nous voulons encore plus de contrôle, nous pouvons créer notre propre classe de délégué et la définir sur les vues qui l'utiliseront. La boîte de dialogue Track Editor illustrée en Figure 10.15 est basée sur un délégué personnalisé. Elle affiche les titres des pistes de musique ainsi que leur durée. Les données stockées dans le modèle seront simplement des `QString` (pour les titres) et des `int` (pour les secondes), mais les durées seront divisées en minutes et en secondes et pourront être modifiées à l'aide de `QTimeEdit`.

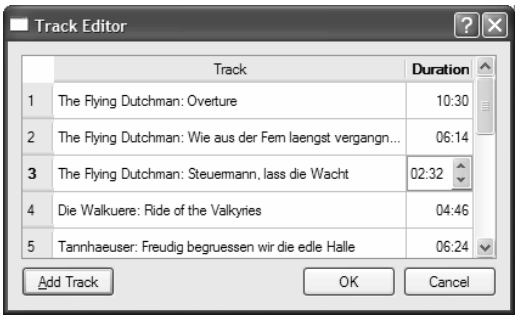


Figure 10.15
La boîte de dialogue Track Editor

La boîte de dialogue Track Editor se sert d'un `QTableWidget`, une sous-classe dédiée à l'affichage d'éléments qui agit sur des `QTableWidgetItem`. Les données sont proposées sous forme d'une liste de `Track` :

```
class Track
{
public:
   Track(const QString &title = "", int duration = 0);

   QString title;
   int duration;
};
```

Voici un extrait du constructeur qui présente la création et l'alimentation en données du widget tableau :

```
TrackEditor::TrackEditor(QList<Track> *tracks, QWidget *parent)
   : QDialog(parent)
{
   this->tracks = tracks;

   tableWidget = new QTableWidget(tracks->count(), 2);
   tableWidget->setItemDelegate(new TrackDelegate(1));
   tableWidget->setHorizontalHeaderLabels(
         QStringList() << tr("Track") << tr("Duration"));

   for (int row = 0; row < tracks->count(); ++row) {
      Track track = tracks->at(row);

      QTableWidgetItem *item0 = new QTableWidgetItem(track.title);
      tableWidget->setItem(row, 0, item0);

      QTableWidgetItem *item1
         = new QTableWidgetItem(QString::number(track.duration));
      item1->setTextAlignment(Qt::AlignRight);
      tableWidget->setItem(row, 1, item1);
   }
   ...
}
```

Le constructeur crée un widget tableau et au lieu d'utiliser simplement le délégué par défaut, nous définissons notre `TrackDelegate` personnalisé, lui transmettant la colonne qui contient les données de temps. Nous configurons d'abord les en-têtes des colonnes, puis nous parcourons les données, en alimentant les lignes avec le nom et la durée de chaque piste.

Le reste du constructeur et de la boîte de dialogue TrackEditor ne présentent aucune particularité, nous allons donc analyser maintenant le `TrackDelegate` qui gère le rendu et la modification des données de la piste.

```
class TrackDelegate : public QItemDelegate
{
   Q_OBJECT

public:
   TrackDelegate(int durationColumn, QObject *parent = 0);
   void paint(QPainter *painter, const QStyleOptionViewItem &option,
              const QModelIndex &index) const;
   QWidget *createEditor(QWidget *parent,
                         const QStyleOptionViewItem &option,
                         const QModelIndex &index) const;
   void setEditorData(QWidget *editor, const QModelIndex &index) const;
   void setModelData(QWidget *editor, QAbstractItemModel *model,
                     const QModelIndex &index) const;

private slots:
   void commitAndCloseEditor();

private:
   int durationColumn;
};
```

Nous utilisons `QItemDelegate` comme classe de base afin de bénéficier de l'implémentation du délégué par défaut. Nous aurions aussi pu utiliser `QAbstractItemDelegate` si nous avions voulu tout commencer à zéro. Pour proposer un délégué qui peut modifier des données, nous devons implémenter `createEditor()`, `setEditorData()` et `setModelData()`. Nous implémentons aussi `paint()` pour modifier l'affichage de la colonne de durée.

```
TrackDelegate::TrackDelegate(int durationColumn, QObject *parent)
   : QItemDelegate(parent)
{
   this->durationColumn = durationColumn;
}
```

Le paramètre `durationColumn` du constructeur indique au délégué quelle colonne contient la durée de la piste.

```
void TrackDelegate::paint(QPainter *painter,
                          const QStyleOptionViewItem &option,
                          const QModelIndex &index) const
{
   if (index.column() == durationColumn) {
      int secs = index.model()->data(index, Qt::DisplayRole).toInt();
```

```
        QString text = QString("%1:%2")
                       .arg(secs / 60, 2, 10, QChar('0'))
                       .arg(secs % 60, 2, 10, QChar('0'));

        QStyleOptionViewItem myOption = option;
        myOption.displayAlignment = Qt::AlignRight | Qt::AlignVCenter;

        drawDisplay(painter, myOption, myOption.rect, text);
        drawFocus(painter, myOption, myOption.rect);
    } else{
        QItemDelegate::paint(painter, option, index);
    }
}
```

Vu que nous voulons afficher la durée sous la forme "*minutes : secondes*" nous avons réimplémenté la fonction `paint()`. Les appels de `arg()` reçoivent un nombre entier à afficher sous forme de chaîne, la quantité de caractères que la chaîne doit contenir, la base de l'entier (10 pour un nombre décimal) et le caractère de remplissage.

Pour justifier le texte à droite, nous copions les options de style en cours et nous remplaçons l'alignement par défaut. Nous appelons ensuite `QItemDelegate::drawDisplay()` pour dessiner le texte, suivi de `QItemDelegate::drawFocus()` qui tracera un rectangle de focus si l'élément est actif et ne fera rien dans les autres cas. La fonction `drawDisplay()` se révèle très pratique, notamment avec nos propres options de style. Nous pourrions aussi dessiner directement en utilisant le painter.

```
QWidget *TrackDelegate::createEditor(QWidget *parent,
        const QStyleOptionViewItem &option,
        const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        QTimeEdit *timeEdit = new QTimeEdit(parent);
        timeEdit->setDisplayFormat("mm:ss");
        connect(timeEdit, SIGNAL(editingFinished()),
                this, SLOT(commitAndCloseEditor()));
        return timeEdit;
    } else {
        return QItemDelegate::createEditor(parent, option, index);
    }
}
```

Nous ne voulons modifier que la durée des pistes, le changement des noms de piste reste à la charge du délégué par défaut. Pour ce faire, nous vérifions pour quelle colonne un éditeur a été demandé au délégué. S'il s'agit de la colonne de durée, nous créons un `QTimeEdit`, nous définissons le format d'affichage de manière appropriée et nous relions son signal `editingFinished()` à notre slot `commitAndCloseEditor()`. Pour toute autre colonne, nous cédons la gestion des modifications au délégué par défaut.

```
void TrackDelegate::commitAndCloseEditor()
{
```

```
    QTimeEdit *editor = qobject_cast<QTimeEdit *>(sender());
    emit commitData(editor);
    emit closeEditor(editor);
}
```

Si l'utilisateur appuie sur Entrée ou déplace le focus hors du `QTimeEdit` (mais pas s'il appuie sur Echap), le signal `editingFinished()` est émis et le slot `commitAndCloseEditor()` est invoqué. Ce slot émet le signal `commitData()` pour informer la vue qu'il y a des données modifiées qui remplacent les données existantes. Il émet aussi le signal `closeEditor()` pour informer la vue que cet éditeur n'est plus requis, le modèle le supprimera donc. L'éditeur est récupéré à l'aide de `QObject::sender()` qui retourne l'objet qui a émis le signal qui a déclenché le slot. Si l'utilisateur annule (en appuyant sur Echap), la vue supprimera simplement l'éditeur.

```
void TrackDelegate::setEditorData(QWidget *editor,
                                  const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        int secs = index.model()->data(index, Qt::DisplayRole).toInt();
        QTimeEdit *timeEdit = qobject_cast<QTimeEdit *>(editor);
        timeEdit->setTime(QTime(0, secs / 60, secs % 60));
    } else {
        QItemDelegate::setEditorData(editor, index);
    }
}
```

Quand l'utilisateur initie une modification, la vue appelle `createEditor()` pour créer un éditeur, puis `setEditorData()` pour initialiser l'éditeur avec les données en cours de l'élément. Si l'éditeur concerne la colonne de durée, nous extrayons la durée de la piste en secondes et nous définissons le temps de `QTimeEdit` avec le nombre correspondant de minutes et de secondes ; sinon nous laissons le délégué par défaut s'occuper de l'initialisation.

```
void TrackDelegate::setModelData(QWidget *editor,
                                 QAbstractItemModel *model,
                                 const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        QTimeEdit *timeEdit = qobject_cast<QTimeEdit *>(editor);
        QTime time = timeEdit->time();
        int secs = (time.minute() * 60) + time.second();
        model->setData(index, secs);
    } else {
        QItemDelegate::setModelData(editor, model, index);
    }
}
```

Si l'utilisateur termine la modification (par exemple en cliquant en dehors du widget ou en appuyant sur Entrée ou Tab) au lieu de l'annuler, le modèle doit être mis à jour avec les données de l'éditeur. Si la durée a changé, nous extrayons les minutes et les secondes du `QTimeEdit` et nous configurons les données avec le nombre équivalent en secondes.

Même si ce n'est pas nécessaire dans ce cas, il est tout à fait possible de créer un délégué personnalisé qui contrôle étroitement la modification et l'affichage de n'importe quel élément d'un modèle. Nous avons choisi de nous occuper d'une colonne particulière, mais vu que `QModelIndex` est transmis à toutes les fonctions de `QItemDelegate` que nous réimplémentons, nous pouvons prendre le contrôle par colonne, ligne, zone rectangulaire, parent ou toute combinaison de ceux-ci jusqu'aux éléments individuels si nécessaire.

Dans ce chapitre, nous vous avons présenté un large aperçu de l'architecture modèle/vue de Qt. Nous avons vu comment utiliser les sous-classes dédiées à l'affichage et les modèles prédéfinis de Qt et comment créer des modèles et des délégués personnalisés. Toutefois, l'architecture modèle/vue est si riche que nous n'aurions pas suffisamment de place pour traiter tous ses aspects. Par exemple, nous pourrions créer une vue personnalisée qui n'affiche pas ses éléments sous forme de liste, de tableau ou d'arborescence. C'est ce que propose l'exemple Chart situé dans le répertoire examples/itemviews/chart de Qt, qui présente une vue personnalisée affichant des données du modèle sous forme de graphique à secteurs.

Il est également possible d'employer plusieurs vues pour afficher le même modèle sans mise en forme. Toute modification effectuée *via* une vue se reflétera automatiquement et immédiatement dans les autres vues. Ce type de fonctionnalité est particulièrement utile pour afficher de grands ensembles de données où l'utilisateur veut voir des sections de données qui sont logiquement éloignées les unes des autres. L'architecture prend en charge les sélections : quand deux vues ou plus utilisent le même modèle, chaque vue peut être définie de manière à avoir ses propres sélections indépendantes, ou alors les sélections peuvent se répartir entre les vues.

La documentation en ligne de Qt aborde la programmation d'affichage d'éléments et les classes qui l'implémentent. Consultez le site **http://doc.trolltech.com/4.1/model-view.html** pour obtenir une liste des classes pertinentes et **http://doc.trolltech.com/4.1/model-view-programming.html** pour des informations supplémentaires et des liens vers les exemples fournis avec Qt.

11

Classes conteneur

Au sommaire de ce chapitre

- ✔ Conteneurs séquentiels
- ✔ Conteneurs associatifs
- ✔ Algorithmes génériques
- ✔ Chaînes, tableaux d'octets et variants

Les classes conteneur sont des classes template polyvalentes qui stockent des éléments d'un type donné en mémoire. C++ offre déjà de nombreux conteneurs dans la STL (*Standard Template Library*), qui est incluse dans la bibliothèque C++ standard.

Qt fournissant ses propres classes conteneur, nous pouvons utiliser à la fois les conteneurs STL et Qt pour les programmes Qt. Les conteneurs Qt présentent l'avantage de se comporter de la même façon sur toutes les plates-formes et d'être partagés implicitement. Le partage implicite, ou la technique de "copie à l'écriture", est une optimisation qui permet la transmission de conteneurs entiers comme valeurs sans coût significatif pour les performances. Les conteneurs Qt comportent également des classes d'itérateurs simple d'emploi inspirées par Java. Elles peuvent être diffusées au moyen d'un `QDataStream` et elles nécessitent moins de code dans l'exécutable que les conteneurs STL correspondants. Enfin, sur certaines plates-formes matérielles supportées par Qtopia Core (la version Qt pour périphériques mobiles), les conteneurs Qt sont les seuls disponibles.

Qt offre à la fois des conteneurs séquentiels tels que `QVector<T>`, `QLinkedList<T>` et `QList<T>` et des conteneurs associatifs comme `QMap<K,T>` et `QHash<K,T>`. Logiquement, les conteneurs séquentiels stockent les éléments les uns après les autres, alors que les conteneurs associatifs stockent des paires clé/valeur.

Qt fournit également des algorithmes génériques qui réalisent des opérations sur les conteneurs. Par exemple, l'algorithme `qSort()` trie un conteneur séquentiel et `qBinaryFind()` effectue une recherche binaire sur un conteneur séquentiel trié. Ces algorithmes sont similaires à ceux offerts par la STL.

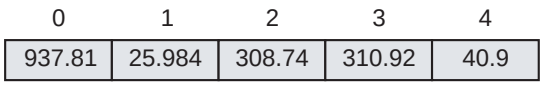
Si vous êtes déjà familier avec les conteneurs de la STL et si vous disposez de cette bibliothèque sur vos plates-formes cibles, vous pouvez les utiliser à la place ou en plus des conteneurs Qt. Pour plus d'informations au sujet des fonctions et des classes de la STL, rendez-vous sur le site Web de SGI à l'adresse **http://www.sgi.com/tech/stl/**.

Dans ce chapitre, nous étudierons également les classes `QString`, `QByteArray` et `QVariant`, qui ont toutes de nombreux points en commun avec les conteneurs. `QString` est une chaîne Unicode 16 bits utilisée dans l'API de Qt. `QByteArray` est un tableau de caractères de 8 bits utilisé pour stocker des données binaires brutes. `QVariant` est un type susceptible de stocker la plupart des types de valeurs Qt et C++.

Conteneurs séquentiels

Un `QVector<T>` est une structure de données de type tableau qui stocke ses éléments à des emplacements adjacents en mémoire. Un vecteur se distingue d'un tableau C++ brut par le fait qu'il connaît sa propre taille et peut être redimensionné. L'ajout d'éléments supplémentaires à la fin d'un vecteur est assez efficace, alors que l'insertion d'éléments devant ou au milieu de celui-ci peut s'avérer coûteux. (voir Figure 11.1)

Figure 11.1
Un vecteur d'éléments de type double



Si nous savons à l'avance combien d'éléments nous seront nécessaires, nous pouvons attribuer au vecteur une taille initiale lors de sa définition et utiliser l'opérateur `[]` pour affecter une valeur aux éléments. Dans le cas contraire, nous devons redimensionner le vecteur ultérieurement ou ajouter les éléments. Voici un exemple dans lequel nous spécifions la taille initiale :

```
QVector<double> vect(3);
vect[0] = 1.0;
vect[1] = 0.540302;
vect[2] = -0.416147;
```

Voici le même exemple, commençant cette fois avec un vecteur vide et utilisant la fonction `append()` pour ajouter des éléments à la fin :

```
QVector<double> vect;
vect.append(1.0);
vect.append(0.540302);
vect.append(-0.416147);
```

Nous pouvons également remplacer `append()` par l'opérateur `<<` :

```
vect << 1.0 << 0.540302 << -0.416147;
```

Vous parcourez les éléments du vecteur à l'aide de `[]` et `count()` :

```
double sum = 0.0;
for (int i = 0; i < vect.count(); ++i)
   sum += vect[i];
```

Les entrées de vecteur créées sans qu'une valeur explicite ne leur soit attribuée sont initialisées au moyen du constructeur par défaut de la classe de l'élément. Les types de base et les types pointeur sont initialisés en zéro.

L'insertion d'éléments au début ou au milieu d'un `QVector<T>`, ou la suppression d'éléments à ces emplacements, risque de ne pas être efficace pour de gros vecteurs. C'est pourquoi Qt offre également `QLinkedList<T>`, une structure de données qui stocke ses éléments à des emplacements non adjacents en mémoire. Contrairement aux vecteurs, les listes chaînées ne prennent pas en charge l'accès aléatoire, mais elles garantissent les performances des insertions et des suppressions. (Voir Figure 11.2)

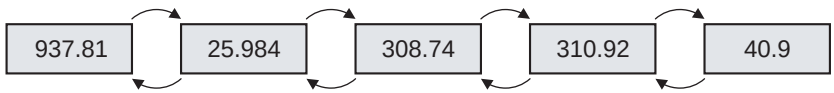


Figure 11.2
Une liste chaînée d'éléments de type double

Les listes chaînées ne fournissent pas l'opérateur `[]`. Il est donc nécessaire de recourir aux itérateurs pour parcourir leurs éléments. Les itérateurs sont également utilisés pour spécifier la position des éléments. Par exemple, le code suivant insère la chaîne "Tote Hosen" entre "Clash" et "Ramones" :

```
QLinkedList<QString> list;
list.append("Clash");
list.append("Ramones");

QLinkedList<QString>::iterator i = list.find("Ramones");
list.insert(i, "Tote Hosen");
```

Nous étudierons les itérateurs en détail ultérieurement dans cette section.

Le conteneur séquentiel `QList<T>` est une "liste-tableau" qui combine les principaux avantages de `QVector<T>` et de `QLinkedList<T>` dans une seule classe. Il prend en charge l'accès aléatoire et son interface est basée sur les index, de la même façon que celle de `QVector`. L'ajout ou la suppression d'un élément à une extrémité d'un `QList<T>` est très rapide. En outre, une insertion au sein d'une liste contenant jusqu'à un millier d'éléments est également très simple. A moins que nous souhaitions réaliser des insertions au milieu de listes de taille très importante ou que nous ayons besoin que les éléments de la liste occupent des adresses consécutives en mémoire, `QList<T>` constitue généralement la classe conteneur polyvalente la plus appropriée. La classe `QStringList` est une sous-classe de `QList<QString>` qui est largement utilisée dans l'API de Qt. En plus des fonctions qu'elle hérite de sa classe de base, elle fournit des fonctions supplémentaires qui la rendent plus souple d'emploi pour la gestion de chaîne. `QStringList` est étudiée dans la dernière section de ce chapitre.

`QStack<T>` et `QQueue<T>` sont deux exemples supplémentaires de sous-classes utilitaires. `QStack<T>` est un vecteur qui fournit `push()`, `pop()` et `top()`. `QQueue<T>` est une liste qui fournit `enqueue()`, `dequeue()` et `head()`.

Pour toutes les classes conteneur rencontrées jusqu'à présent, le type de valeur `T` peut être un type de base tel que `int` ou `double`, un type pointeur ou une classe qui possède un constructeur par défaut (un constructeur qui ne reçoit aucun argument), un constructeur de copie et un opérateur d'affectation. Les classes qui remplissent les conditions requises incluent `QByteArray`, `QDateTime`, `QRegExp`, `QString` et `QVariant`. Les classes Qt qui héritent de `QObject` s'avèrent inadéquates, car il leur manque un constructeur de copie et un opérateur d'affectation. Ceci ne pose pas de problème dans la pratique, car nous pouvons simplement stocker des pointeurs vers des types `QObject` plutôt que les objets eux-mêmes.

Le type de valeur `T` peut également être un conteneur, auquel cas nous devons séparer deux crochets consécutifs par des espaces. Sinon, le compilateur butera sur ce qu'il pense être un opérateur >>. Par exemple :

```
QList<QVector<double> > list;
```

En plus des types que nous venons de mentionner, le type de valeur d'un conteneur peut être toute classe personnalisée correspondant aux critères décrits précédemment. Voici un exemple de classe de ce type :

```
class Movie
{
public:
    Movie(const QString &title = "", int duration = 0);

    void setTitle(const QString &title) { myTitle = title; }
    QString title() const { return myTitle; }
    void setDuration(int duration) { myDuration = duration; }
    QString duration() const { return myDuration; }
```

```
private:
    QString myTitle;
    int myDuration;
};
```

La classe possède un constructeur qui n'exige aucun argument (bien qu'il puisse en recevoir jusqu'à deux). Elle possède également un constructeur de copie et un opérateur d'affectation, tous deux étant implicitement fournis par C++. Pour cette classe, la copie au membre par membre est suffisante. Il n'est donc pas nécessaire d'implémenter votre propre constructeur de copie et votre opérateur d'affectation.

Qt fournit deux catégories d'itérateurs afin de parcourir les éléments stockés dans un conteneur. Les itérateurs de style Java et ceux de style STL. Les itérateurs de style Java sont plus faciles à utiliser, alors que ceux de style STL sont plus puissants et peuvent être combinés avec les algorithmes génériques de Qt et de STL.

Pour chaque classe conteneur, il existe deux types d'itérateurs de style Java : un itérateur en lecture seulement et un itérateur en lecture-écriture. Les classes d'itérateur en lecture seulement sont `QVectorIterator<T>`, `QLinkedListIterator<T>` et `QListIterator<T>`. Les itérateurs en lecture/écriture correspondants comportent le terme `Mutable` dans leur nom (par exemple, `QMutableVectorIterator<T>`). Dans cette discussion, nous allons surtout étudier les itérateurs de `QList` ; les itérateurs pour les listes chaînées et les vecteurs possèdent la même API. (Voir Figure 11.3)

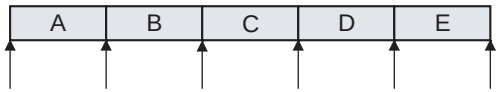


Figure 11.3
Emplacements valides pour les itérateurs de style Java

Le premier point à garder à l'esprit lors de l'utilisation d'itérateurs de style Java est qu'ils ne pointent pas directement vers des éléments. Ils peuvent être situés avant le premier élément, après le dernier ou entre deux. Voici la syntaxe d'une boucle d'itération typique :

```
QList<double> list;
...
QListIterator<double> i(list);
while (i.hasNext()) {
    do_something(i.next());
}
```

L'itérateur est initialisé avec le conteneur à parcourir. A ce stade, l'itérateur est situé juste avant le premier élément. L'appel à `hasNext()` retourne `true` si un élément se situe sur la droite de l'itérateur. La fonction `next()` retourne l'élément situé sur la droite de l'itérateur et avance ce dernier jusqu'à la prochaine position valide.

L'itération vers l'arrière est similaire, si ce n'est que nous devons tout d'abord appeler `toBack()` pour placer l'itérateur après le dernier élément :

```
QListIterator<double> i(list);
i.toBack();
while (i.hasPrevious()) {
    do_something(i.previous());
}
```

La fonction `hasPrevious()` retourne `true` si un élément se trouve sur la gauche de l'itérateur ; `previous()` retourne cet élément et le déplace vers l'arrière. Les itérateurs `next()` et `previous()` retournent l'élément que l'itérateur vient de passer. (Voir Figure 11.4).

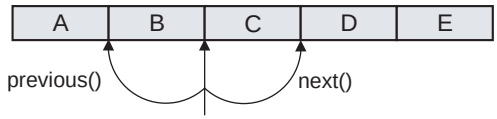


Figure 11.4
Effet de `previous()` *et de* `next()` *sur un itérateur de style Java*

Les itérateurs mutables fournissent des fonctions destinées à insérer, modifier et supprimer des éléments lors de l'itération. La boucle suivante supprime tous les nombres négatifs d'une liste :

```
QMutableListIterator<double> i(list);
while (i.hasNext()) {
    if (i.next() < 0.0)
        i.remove();
}
```

La fonction `remove()` opère toujours sur le dernier élément passé. Elle fonctionne également lors de l'itération vers l'arrière.

```
QMutableListIterator<double> i(list);
i.toBack();
while (i.hasPrevious()) {
    if (i.previous() < 0.0)
        i.remove();
}
```

De la même façon, les itérateurs mutables de style Java fournissent une fonction `setValue()` qui modifie le dernier élément passé. Voici comment nous remplacerions des nombres négatifs par leur valeur absolue :

```
QMutableListIterator<double> i(list);
while (i.hasNext()) {
    int val = i.next();
    if (val < 0.0)
        i.setValue(-val);
}
```

Il est également possible d'insérer un élément à l'emplacement courant de l'itérateur en appelant `insert()`. L'itérateur est alors avancé à l'emplacement se situant entre le nouvel élément et l'élément suivant.

En plus des itérateurs de style Java, chaque classe conteneur séquentiel `C<T>` possède deux types d'itérateurs de style STL : `C<T>::iterator` et `C<T>::const_iterator`. La différence entre les deux est que `const_iterator` ne nous permet pas de modifier les données.

La fonction `begin()` d'un conteneur retourne un itérateur de style STL faisant référence au premier élément du conteneur (par exemple `list[0]`), alors que `end()` retourne un itérateur pointant vers l'élément suivant le dernier (par exemple, `list[5]` pour une liste de taille 5). Si un conteneur est vide, `begin()` est égal à `end()`. Cette caractéristique peut être utilisée pour déterminer si le conteneur comporte des éléments, bien qu'il soit généralement plus approprié d'appeler `isEmpty()` à cette fin. (Voir Figure 11.5)

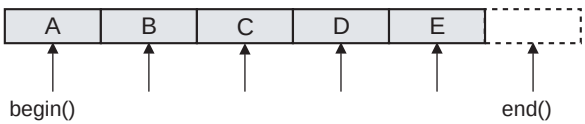


Figure 11.5
Emplacements valides pour les itérateurs de style STL

La syntaxe d'un itérateur de style STL est modelée sur celle des pointeurs C++ dans un tableau. Nous pouvons utiliser les opérateurs `++` et `--` pour passer à l'élément précédent ou suivant et l'opérateur `*` unaire pour récupérer l'élément en cours. Pour `QVector<T>`, l'itérateur et les types `const_iterator` sont simplement des typedefs de `T*` et `constT*`. (Ceci est possible parce que `QVector<T>` stocke ses éléments dans des emplacements consécutifs en mémoire.)

L'exemple suivant remplace chaque valeur d'un `QList<double>` par sa valeur absolue :

```
QList<double>::iterator i = list.begin();
while (i != list.end()) {
    *i = qAbs(*i);
    ++i;
}
```

Quelques fonctions Qt retournent un conteneur. Si nous voulons parcourir la valeur de retour d'une fonction au moyen d'un itérateur de style STL, nous devons prendre une copie du conteneur et parcourir cette copie. Le code suivant, par exemple, illustre comment parcourir correctement le `QList<int>` retourné par `QSplitter ::sizes()` :

```
QList<int> list = splitter->sizes();
QList<int>::const_iterator i = list.begin();
while (i != list.end()) {
    do_something(*i);
    ++i;
}
```

Le code suivant est incorrect :

```
// INEXACT
QList<int>::const_iterator i = splitter->sizes().begin();
while (i != splitter->sizes().end()) {
    do_something(*i);
    ++i;
}
```

En effet, `QSplitter::sizes()` retourne un nouveau `QList<int>` par valeur à chacun de ses appels. Si nous ne stockons pas la valeur de retour, C++ la détruit automatiquement avant même que nous ayons débuté l'itération, nous laissant avec un itérateur sans liaison. Pire encore, à chaque exécution de la boucle, `QSplitter::sizes()` doit générer une nouvelle copie de la liste à cause de l'appel `splitter_>sizes().end()`.

En résumé : lorsque vous utilisez des itérateurs de style STL, parcourez toujours vos éléments sur une copie d'un conteneur.

Avec les itérateurs de style Java en lecture seulement, il est inutile de recourir à une copie. L'itérateur se charge de créer cette copie en arrière-plan. Par exemple :

```
QListIterator<int> i(splitter->sizes());
while (i.hasNext()) {
    do_something(i.next());
}
```

La copie d'un conteneur tel que celui-ci semble coûteuse, mais il n'en est rien, grâce à l'optimisation obtenue par le *partage implicite*. La copie d'un conteneur Qt est pratiquement aussi rapide que celle d'un pointeur unique. Les données ne sont véritablement copiées que si l'une des copies est changée – et tout ceci est géré automatiquement à l'arrière-plan. C'est pourquoi le partage implicite est quelquefois nommé "copie à l'écriture".

L'intérêt du partage implicite est qu'il s'agit d'une optimisation dont nous bénéficions sans intervention de la part du programmeur. En outre, le partage implicite favorise un style de programmation clair, où les objets sont retournés par valeur. Considérez la fonction suivante :

```
QVector<double> sineTable()
{
    QVector<double> vect(360);
    for (int i = 0; i < 360; ++i)
        vect[i] = sin(i / (2 * M_PI));
    return vect;
}
```

Voici l'appel à la fonction :

```
QVector<double> table = sineTable();
```

STL, nous incite plutôt à transmettre le vecteur comme référence non const pour éviter l'exécution de la copie lorsque la valeur de retour de la fonction est stockée dans une variable :

```
using namespace std;

void sineTable(vector<double> &vect)
{
    vect.resize(360);
    for (int i = 0; i < 360; ++i)
        vect[i] = sin(i / (2 * M_PI));
}
```

L'appel devient alors plus difficile à écrire et à lire :

```
vector<double> table;
sineTable(table);
```

Qt utilise le partage implicite pour tous ses conteneurs ainsi que pour de nombreuses autres classes, dont `QByteArray`, `QBrush`, `QFont`, `QImage`, `QPixmap` et `QString` .

Le partage implicite est une garantie de la part de Qt que les données ne seront pas copiées si nous ne les modifions pas. Pour obtenir le meilleur du partage implicite, nous pouvons adopter deux nouvelles habitudes de programmation. L'une consiste à coder la fonction `at()` au lieu de l'opérateur `[]` pour un accès en lecture seulement sur une liste ou un vecteur (non const). Les conteneurs Qt ne pouvant pas déterminer si `[]` apparaît sur le côté gauche d'une affectation ou non, le pire est envisagé et une copie intégrale est déclenchée – alors que `at()` n'est pas autorisé sur le côté gauche d'une affectation.

Un problème similaire se pose lorsque nous parcourons un conteneur avec des itérateurs de style STL. Dès que nous appelons `begin()` ou `end()` sur un conteneur non const, Qt force une copie complète si les données sont partagées. Pour éviter ceci, la solution consiste à utiliser `const_iterator`, `constBegin()` et `constEnd()` dès que possible.

Qt fournit une dernière méthode pour parcourir les éléments situés dans un conteneur séquentiel : la boucle `foreach`. Voici sa syntaxe :

```
QLinkedList<Movie> list;
...
foreach (Movie movie, list) {
    if (movie.title() == "Citizen Kane") {
        cout << "Found Citizen Kane" << endl;
        break;
    }
}
```

Le pseudo mot-clé `foreach` est implémenté sous la forme d'une boucle `for` standard. A chaque itération de la boucle, la variable d'itération (`movie`) est définie en un nouvel élément, commençant au premier élément du conteneur et progressant vers l'avant. La boucle `foreach` reçoit automatiquement une copie du conteneur. Elle ne sera donc pas affectée si le conteneur est modifié durant l'itération.

Fonctionnement du partage implicite

Le partage implicite s'effectue automatiquement en arrière-plan. Aucune action n'est donc nécessaire dans notre code pour que cette optimisation se produise. Mais comme il est intéressant de comprendre comment les choses fonctionnent, nous allons étudier un exemple et voir ce qui se passe en interne. L'exemple utilise `QString`, une des nombreuses classes implicitement partagées de Qt.

```
QString str1 = "Humpty";
QString str2 = str1;
```

Nous définissons `str1` en `"Humpty"` et `str2` de sorte qu'il soit égal à `str1`. A ce stade, les deux objets `QString` pointent vers la même structure de données interne en mémoire. Avec les données de type caractère, il existe pour une structure de données un compteur de référence indiquant le nombre de `QString` pointant vers celle-ci. `str1` et `str2` pointant vers la même donnée, le compteur de référence indique `2`.

```
str2[0] = 'D';
```

Lorsque nous modifions `str2`, il réalise tout d'abord une copie intégrale des données pour s'assurer que `str1` et `str2` pointent vers des structures de données différentes, puis il applique la modification à sa propre copie des données. Le compteur de référence des données de `str1` (`"Humpty"`) indique alors `1` et celui des données de `str2` (`"Dumpty"`) est défini en `1`. Quand un compteur de référence indique `1`, les données ne sont pas partagées.

```
str2.truncate(4);
```

Si nous modifions de nouveau `str2`, aucune copie ne se produit car le compteur de référence des données de `str2` indique 1. La fonction `truncate()` agit directement sur les données de `str2`, résultant en la chaîne `"Dump"`. Le compteur de référence reste à `1`.

```
str1 = str2;
```

Lorsque nous affectons `str2` à `str1`, le compteur de référence des données de `str1` descend à `0`, ce qui signifie qu'aucun `QString` n'utilise plus la donnée `"Humpty"`. La donnée est alors libérée de la mémoire. Les deux `QString` pointent vers `"Dump"`, dont le compteur de référence indique maintenant `2`.

Le partage de données est une option souvent ignorée dans les programmes multithread, à cause des conditions de compétition dans le décompte des références. Avec Qt, ceci n'est plus un problème. En interne, les classes conteneur utilisent des instructions du langage d'assembly pour effectuer un décompte de références atomique. Cette technologie est à la portée des utilisateurs de Qt par le biais des classes `QSharedData` et `QSharedDataPointer`.

Les instructions de boucle `break` et `continue` sont prises en charge. Si le corps est constitué d'une seule instruction, les accolades ne sont pas nécessaires. Comme pour une instruction `for`, la variable d'itération peut être définie à l'extérieur de la boucle, comme suit :

```
QLinkedList<Movie> list;
Movie movie;
...
foreach (movie, list) {
    if (movie.title() == "Citizen Kane") {
        cout << "Found Citizen Kane" << endl;
        break;
    }
}
```

La définition de la variable d'itération à l'extérieur de la boucle est la seule solution pour les conteneurs comportant des types de données avec une virgule (par exemple, `QPair<QString,int>`).

Conteneurs associatifs

Un conteneur associatif comporte un nombre arbitraire d'éléments du même type, indexés par une clé. Qt fournit deux classes de conteneurs associatifs principales : `QMap<K,T>` et `QHash<K,T>`.

Un `QMap<K,T>` est une structure de données qui stocke des paires clé/valeur dans un ordre croissant des clés. Cette organisation permet d'obtenir de bonnes performances en matière de recherche et d'insertion ainsi qu'une itération ordonnée. En interne, `QMap<K,T>` est implémenté sous forme de liste à branchement. (Voir Figure 11.6)

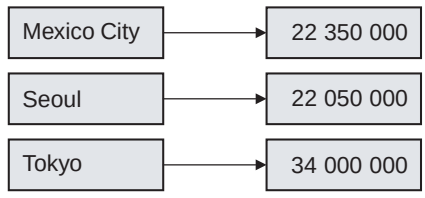


Figure 11.6
Un map de `QString` *vers* `int`

Un moyen simple d'insérer des éléments dans un map consiste à appeler `insert()` :

```
QMap<QString, int> map;
map.insert("eins", 1);
map.insert("sieben", 7);
map.insert("dreiundzwanzig", 23);
```

Nous avons aussi la possibilité d'affecter simplement une valeur à une clé donnée comme suit :

```
map["eins"] = 1;
map["sieben"] = 7;
map["dreiundzwanzig"] = 23;
```

L'opérateur `[]` peut être utilisé à la fois pour l'insertion et la récupération. Si `[]` est utilisé pour récupérer une valeur d'une clé non existante dans un map non const, un nouvel élément sera créé avec la clé donnée et une valeur vide. L'utilisation de la fonction `value()` à la place de `[]` pour récupérer les éléments permet d'éviter la création accidentelle de valeurs vides :

```
int val = map.value("dreiundzwanzig");
```

Si la clé n'existe pas, une valeur par défaut est retournée et aucun nouvel élément n'est créé. Pour les types de base et les types pointeur, la valeur retournée est nulle. Nous pouvons spécifier une autre valeur par défaut comme second argument pour `value()` :

```
int int seconds = map.value("delay", 30);
```

Cette ligne de code est équivalente à :

```
int seconds = 30;
if (map.contains("delay"))
    seconds = map.value("delay");
```

Les types de données `K` et `T` d'un `QMap<K,T>` peuvent être des types de base tels que `int` et `double`, des types pointeur ou de classes possédant un constructeur par défaut, un constructeur de copie et un opérateur d'affectation. En outre, le type `K` doit fournir un `operator<()` car `QMap<K,T>` utilise cet opérateur pour stocker les éléments dans un ordre de clé croissant.

`QMap<K,T>` possède deux fonctions utilitaires `keys()` et `values()`, qui s'avèrent particulièrement intéressantes pour travailler avec de petits ensembles de données. Elles retournent des `QList` des clés et valeurs d'un map.

Les maps sont généralement à valeur unique : si une nouvelle valeur est affectée à une clé existante, l'ancienne valeur est remplacée par la nouvelle. De cette façon, deux éléments ne partagent pas la même clé. Il est possible d'avoir des valeurs multiples pour la même clé en utilisant la fonction `insertMulti()` ou la sous-classe utilitaire `QMultiMap<K,T>`. `QMap<K,T>` possède une surcharge `values(constK &)` qui retourne un `QList` de toutes les valeurs pour une clé donnée. Par exemple :

```
QMultiMap<int, QString> multiMap;
multiMap.insert(1, "one");
multiMap.insert(1, "eins");
multiMap.insert(1, "uno");

QList<QString> vals = multiMap.values(1);
```

Un `QHash<K,T>` est une structure de données qui stocke des paires clé/valeur dans une table de hachage. Son interface est pratiquement identique à celle de `QMap<K,T>`, mais ses exigences concernant le type template K sont différentes et il offre des opérations de recherche beaucoup plus rapides que `QMap<K,T>`. Une autre différence est que `QHash<K,T>` n'est pas ordonné.

En complément des conditions standard concernant tout type de valeur stocké dans un conteneur, le type `K` d'un `QHash<K,T>` doit fournir un `operator==()` et être supporté par une

fonction `QHash()` globale qui retourne une valeur de hachage pour une clé. Qt fournit déjà des fonctions `QHash()` pour les types entiers, les types pointeur, `QChar`, `QString` et `QByteArray`.

`QHash<K,T>` alloue automatiquement un nombre principal de blocs pour sa table de hachage interne et redimensionne celle-ci lors de l'insertion ou de la suppression d'éléments. Il est également possible de régler avec précision les performances en appelant `reserve()` pour spécifier le nombre d'éléments à stocker dans la table et `squeeze()` pour réduire cette table en fonction du nombre d'éléments en cours. Une pratique courante consiste à appeler `reserve()` avec le nombre maximum d'éléments susceptibles d'être stockés, puis à insérer les données et finalement à appeler `squeeze()` pour réduire l'utilisation de la mémoire si les éléments sont moins nombreux que prévu.

Les hachages sont généralement à valeur unique, mais plusieurs valeurs peuvent être affectées à la même clé à l'aide de la fonction `insertMulti()` ou de la sous-classe utilitaire `QMultihash<K,T>`.

En plus de `QHash<K,T>`, Qt fournit une classe `QCache<K,T>` qui peut être utilisée pour placer en cache des objets associés à une clé, et un conteneur `QSet<K>` qui ne stocke que des clés. En interne, tous deux reposent sur `QHash<K,T>` et présentent les mêmes exigences concernant le type `K`.

Le moyen le plus simple de parcourir toutes les paires clé/valeur stockées dans un conteneur associatif consiste à utiliser un itérateur de style Java. Les itérateurs de ce style utilisés pour les conteneurs associatifs ne fonctionnent pas tout à fait de la même façon que leurs homologues séquentiels. La principale différence est la suivante : les fonctions `next()` et `previous()` retournent un objet qui représente une paire clé/valeur, et non simplement une valeur. Les composants clé et valeur sont accessibles depuis cet objet en tant que `key()` et `value()`. Par exemple :

```
QMap<QString, int> map;
...
int sum = 0;
QMapIterator<QString, int> i(map);
while (i.hasNext())
    sum += i.next().value();
```

Si nous devons accéder à la fois à la clé et à la valeur, nous pouvons simplement ignorer la valeur de retour de `next()` ou de `previous()` et exécuter les fonctions `key()` et `value()` de l'itérateur, qui opèrent sur le dernier élément franchi :

```
QMapIterator<QString, int> i(map);
while (i.hasNext()) {
    i.next();
    if (i.value() > largestValue) {
        largestKey = i.key();
        largestValue = i.value();
    }
}
```

Les itérateurs mutables possèdent une fonction `setValue()` qui modifie la valeur associée à un élément courant :

```
QMutableMapIterator<QString, int> i(map);
while (i.hasNext()) {
    i.next();
    if (i.value() < 0.0)
        i.setValue(-i.value());
}
```

Les itérateurs de style STL fournissent également des fonctions `key()` et `value()`. Avec des types d'itérateur non const, `value()` retourne une référence non const, ce qui nous permet de changer la valeur au cours de l'itération. Remarquez que, bien que ces itérateurs soient "de style STL", ils présentent des différences significatives avec les itérateurs `map<K,T>` de STL, qui sont basés sur `pair<K,T>`.

La boucle `foreach` fonctionne également avec les conteneurs associatifs, mais uniquement avec le composant valeur des paires clé/valeur. Si nous avons besoin des deux composants clé et valeur des éléments, nous pouvons appeler les fonctions `keys()` et `values(constK &)` dans des boucles `foreach` imbriquées comme suit :

```
QMultiMap<QString, int> map;
...
foreach (QString key, map.keys()) {
    foreach (int value, map.values(key)) {
        do_something(key, value);
    }
}
```

Algorithmes génériques

L'en-tête `<QtAlgorithms>` déclare un ensemble de fonctions template globales qui implémentent des algorithmes de base sur les conteneurs. La plupart de ces fonctions agissent sur des itérateurs de style STL.

L'en-tête STL `<algorithm>` fournit un ensemble d'algorithmes génériques plus complet. Ces algorithmes peuvent être employés avec des conteneurs Qt ainsi que des conteneurs STL. Si les implémentations STL sont disponibles sur toutes vos plates-formes, il n'y a probablement aucune raison de ne pas utiliser les algorithmes STL lorsque Qt ne propose pas l'algorithme équivalent. Nous présenterons ici les algorithmes Qt les plus importants.

L'algorithme `qFind()` recher286che une valeur particulière dans un conteneur. Il reçoit un itérateur "begin" et un itérateur "end" et retourne un itérateur pointant sur le premier élément correspondant, ou "end" s'il n'existe pas de correspondance. Dans l'exemple suivant, `i` est défini en `list.begin() + 1` alors que `j` est défini en `list.end()`.

```
QStringList list;
```

```
list << "Emma" << "Karl" << "James" << "Mariette";

QStringList::iterator i = qFind(list.begin(), list.end(), "Karl");
QStringList::iterator j = qFind(list.begin(), list.end(), "Petra");
```

L'algorithme `qBinaryFind()` effectue une recherche similaire à `qFind()`, mais il suppose que les éléments sont triés dans un ordre croissant et utilise la recherche binaire rapide au lieu de la recherche linéaire de `qFind()`.

L'algorithme `qFill()` remplit un conteneur avec une valeur particulière :

```
QLinkedList<int> list(10);
qFill(list.begin(), list.end(), 1009);
```

Comme les autres algorithmes basés sur les itérateurs, nous pouvons également utiliser `qFill()` sur une partie du conteneur en variant les arguments. L'extrait de code suivant initialise les cinq premiers éléments d'un vecteur en 1009 et les cinq derniers éléments en 2013 :

```
QVector<int> vect(10);
qFill(vect.begin(), vect.begin() + 5, 1009);
qFill(vect.end() - 5, vect.end(), 2013);
```

L'algorithme `qCopy()` copie des valeurs d'un conteneur à un autre :

```
QVector<int> vect(list.count());
qCopy(list.begin(), list.end(), vect.begin());
```

`qCopy()` peut également être utilisé pour copier des valeurs dans le même conteneur, ceci tant que la plage source et la plage cible ne se chevauchent pas. Dans l'extrait de code suivant, nous l'utilisons pour remplacer les deux derniers éléments d'une liste par les deux premiers :

```
qCopy(list.begin(), list.begin() + 2, list.end() - 2);
```

L'algorithme `qSort()` trie les éléments du conteneur en ordre croissant :

```
qSort(list.begin(), list.end());
```

Par défaut, `qSort()` se sert de l'opérateur `<` pour comparer les éléments. Pour trier les éléments en ordre décroissant, transmettez `qGreater<T>()` comme troisième argument (où `T` est le type des valeurs du conteneur), comme suit :

```
qSort(list.begin(), list.end(), qGreater<int>());
```

Nous pouvons utiliser le troisième paramètre pour définir un critère de tri personnalisé. Voici, par exemple, une fonction de comparaison "inférieur à" qui compare des `QString` sans prendre la casse (majuscule-minuscule) en considération :

```
bool insensitiveLessThan(const QString &str1, const QString &str2)
{
    return str1.toLower() < str2.toLower();
```

```
}
```

L'appel à `qSort()` devient alors :

```
QStringList list;
...
qSort(list.begin(), list.end(), insensitiveLessThan);
```

L'algorithme `qStableSort()` est similaire à `qSort()`, si ce n'est qu'il garantit que les éléments égaux apparaissent dans le même ordre avant et après le tri. Cette caractéristique est utile si le critère de tri ne prend en compte que des parties de la valeur et si les résultats sont visibles pour l'utilisateur. Nous avons utilisé `qStableSort()` dans le Chapitre 4 pour implémenter le tri dans l'application Spreadsheet.

L'algorithme `qDeleteAll()` appelle `delete` sur chaque pointeur stocké dans un conteneur. Ceci n'est utile que pour les conteneurs dont le type de valeur est un type pointeur. Après l'appel, les éléments sont toujours présents, vous exécutez `clear()` sur le conteneur. Par exemple :

```
qDeleteAll(list);
list.clear();
```

L'algorithme `qSwap()` échange la valeur de deux variables. Par exemple :

```
int x1 = line.x1();
int x2 = line.x2();
if (x1 > x2)
    qSwap(x1, x2);
```

Enfin, l'en-tête `<QtGlobal>` fournit plusieurs définitions utiles, dont la fonction `qAbs()`, qui retourne la valeur absolue de son argument ainsi que les fonctions `qMin()` et `qMax()` qui retournent le minimum ou le maximum entre deux valeurs.

Chaînes, tableaux d'octets et variants

`QString`, `QByteArray` et `QVariant` sont trois classes ayant de nombreux points en commun avec les conteneurs. Elles sont susceptibles d'être utilisées à la place de ceux-ci dans certaines situations. En outre, comme les conteneurs, ces classes utilisent le partage implicite pour optimiser la mémoire et la vitesse.

Nous allons commencer par `QString`. Les chaînes sont utilisées par tout programme GUI, non seulement pour l'interface utilisateur, mais également en tant que structures de données. C++ fournit en natif deux types de chaînes : les traditionnels tableaux de caractères terminés par "0" et la classe `std::string`. Contrairement à celles-ci, `QString` contient des valeurs Unicode 16 bits. Unicode comprend les systèmes ASCII et Latin-1, avec leurs valeurs numériques habituelles. Mais `QString` étant une classe 16 bits, elle peut représenter des milliers de caractères différents utilisés dans la plupart des langues mondiales. Reportez-vous au Chapitre 17 pour de plus amples informations concernant Unicode.

Lorsque vous utilisez `QString`, vous n'avez pas besoin de vous préoccuper de détails comme l'allocation d'une mémoire suffisante ou de vérifier que la donnée est terminée par "0". Conceptuellement, les objets `QString` peuvent être considérés comme des vecteurs de `QChar`. Un `QString` peut intégrer des caractères "0". La fonction `length()` retourne la taille de la chaîne entière, dont les caractères "0" intégrés.

`QString` fournit un opérateur + binaire destiné à concaténer deux chaînes ainsi qu'un opérateur += dont la fonction est d'accoler une chaîne à une autre. Voici un exemple combinant + et +=.

```
QString str = "User: ";
str += userName + "\n";
```

Il existe également une fonction `QString::append()` dont la tâche est identique à celle de l'opérateur += :

```
str = "User: ";
str.append(userName);
str.append("\n");
```

Un moyen totalement différent de combiner des chaînes consiste à utiliser la fonction `sprintf()` de `QString` :

```
str.sprintf("%s %.1f%%", "perfect competition", 100.0);
```

Cette fonction prend en charge les mêmes spécificateurs de format que la fonction `sprintf()` de la bibliothèque C++. Dans l'exemple ci-dessus, "perfect competition 100.0 %" est affecté à `str`.

Un moyen supplémentaire de créer une chaîne à partir d'autres chaînes ou de nombres consiste à utiliser `arg()` :

```
str = QString("%1 %2 (%3s-%4s)")
    .arg("permissive").arg("society").arg(1950).arg(1970);
```

Dans cet exemple, "%1", "%2", "%3" et "%4" sont remplacés par "permissive", "society", "1950" et "1970", respectivement. On obtient "permissive society (1950s-1970s)". Il existe des surcharges de `arg()` destinées à gérer divers types de données. Certaines surcharges comportent des paramètres supplémentaires afin de contrôler la largeur de champ, la base numérique ou la précision de la virgule flottante. En général, `arg()` représente une solution bien meilleure que `sprintf()`, car elle est de type sécurisé, prend totalement en charge Unicode et autorise les convertisseurs à réordonner les paramètres "%*n*".

`QString` peut convertir des nombres en chaînes au moyen de la fonction statique `QString::number()` :

```
str = QString::number(59.6);
```

Ou en utilisant la fonction `setNum()` :

```
str.setNum(59.6);
```

La conversion inverse, d'une chaîne en un nombre, est réalisée à l'aide de `toInt()`, `toLongLong()`, `toDouble()` et ainsi de suite. Par exemple :

```
bool ok;
double d = str.toDouble(&ok);
```

Ces fonctions peuvent recevoir en option un pointeur facultatif vers une variable `bool` et elles définissent la variable en `true` ou `false` selon le succès de la conversion. Si la conversion échoue, les fonctions retournent zéro.

Il est souvent nécessaire d'extraire des parties d'une chaîne. La fonction `mid()` retourne la sous-chaîne débutant à un emplacement donné (premier argument) et de longueur donnée (second argument). Par exemple, le code suivant affiche "pays" sur la console[1] :

```
QString str = "polluter pays principle";
qDebug() << str.mid(9, 4);
```

Si nous omettons le second argument, `mid()` retourne la sous-chaîne débutant à l'emplacement donné et se terminant à la fin de la chaîne. Par exemple, le code suivant affiche "pays principle" sur la console :

```
QString str = "polluter pays principle";
qDebug() << str.mid(9);
```

Il existe aussi des fonctions `left()` et `right()` dont la tâche est similaire. Toutes deux reçoivent un nombre de caractères, *n*, et retournent les *n* premiers ou derniers caractères de la chaîne. Par exemple, le code suivant affiche "polluter principle" sur la console :

```
QString str = "polluter pays principle";
qDebug() << str.left(8) << " " << str.right(9);
```

Pour rechercher si un chaîne contient un caractère particulier, une sous-chaîne ou une expression régulière, nous pouvons utiliser l'une des fonctions `indexOf()` de QString :

```
QString str = "the middle bit";
int i = str.indexOf("middle");
```

Dans ce cas, `i` sera défini en `4`. La fonction `indexOf()` retourne `1` en cas d'échec et reçoit en option un emplacement de départ ainsi qu'un indicateur de sensibilité à la casse.

---

1. La syntaxe `qDebug()<<arg` utilisée ici nécessite l'inclusion du fichier d'en-tête `<QtDebug>`, alors que la syntaxe `qDebug("…",arg)` est disponible dans tout fichier incluant au moins un en-tête Qt.

Si nous souhaitons simplement vérifier si une chaîne commence ou se termine par quelque chose, nous pouvons utiliser les fonctions `startsWith()` et `endsWith()` :

```
if (url.startsWith("http:") && url.endsWith(".png"))
    ...
```

Ce qui est à la fois plus rapide et plus simple que :

```
if (url.left(5) == "http:" && url.right(4) == ".png")
    ...
```

La comparaison de chaînes avec l'opérateur == différencie les majuscules des minuscules. Si nous comparons des chaînes visibles pour l'utilisateur, `localeAwareCompare()` représente généralement un bon choix, et si nous souhaitons effectuer des comparaisons sensibles à la casse, nous pouvons utiliser `toUpper()` ou `toLower()`. Par exemple :

```
if (fileName.toLower() == "readme.txt")
    ...
```

Pour remplacer une certaine partie d'une chaîne par une autre chaîne, nous codons `replace()` :

```
QString str = "a cloudy day";
str.replace(2, 6, "sunny");
```

On obtient "a sunny day". Le code peut être réécrit de façon à exécuter `remove()` et `insert()`.

```
str.remove(2, 6);
str.insert(2, "sunny");
```

Dans un premier temps, nous supprimons six caractères en commençant à l'emplacement 2, ce qui aboutit à la chaîne "a day" (avec deux espaces), puis nous insérons "sunny" à ce même emplacement.

Des versions surchargées de `replace()` permettent de remplacer toutes les occurrences de leur premier argument par leur second argument. Par exemple, voici comment remplacer toutes les occurrences de "&" par "&" dans une chaîne :

```
str.replace("&", "&amp;");
```

Il est très souvent nécessaire de supprimer les blancs (tels que les espaces, les tabulations et les retours à la lignes) dans une chaîne. `QString` possède une fonction qui élimine les espaces situés aux deux extrémités d'une chaîne :

```
QString str = " BOB \t THE \nDOG \n";
qDebug() << str.trimmed();
```

La chaîne `str` peut être représentée comme suit :

La chaîne retournée par `trimmed()` est

Lorsque nous gérons les entrées utilisateur, nous avons souvent besoin de remplacer les séquences internes d'un ou de plusieurs caractères d'espace par un espace unique, ainsi que d'éliminer les espaces aux deux extrémités. Voici l'action de la fonction `simplified()` :

```
QString str = " BOB \t THE \nDOG \n";
qDebug() << str.simplified();
```

La chaîne retournée par `simplified()` est :

Une chaîne peut être divisée en un `QStringList` de sous-chaînes au moyen de `QString::split()` :

```
QString str = "polluter pays principle";
QStringList words = str.split(" ");
```

Dans l'exemple ci-dessus, nous divisons la chaîne "polluter pays principle" en trois sous-chaînes : "polluter", "pays" et "principle". La fonction `split()` possède un troisième argument facultatif qui spécifie si les sous-chaînes vides doivent être conservées (option par défaut) ou éliminées.

Les éléments d'un `QStringList` peuvent être unis pour former une chaîne unique au moyen de `join()`. L'argument de `join()` est inséré entre chaque paire de chaînes jointes. Par exemple, voici comment créer une chaîne unique qui est composée de toutes les chaînes contenues dans un `QStringList` triées par ordre alphabétique et séparées par des retours à la lignes :

```
words.sort();
str = words.join("\n");
```

En travaillant avec les chaînes, il est souvent nécessaire de déterminer si elles sont vides ou non. Pour ce faire, appelez `isEmpty()` ou vérifiez si `length()` est égal à `0`.

La conversion de chaînes `const char*` en `QString` est automatique dans la plupart des cas. Par exemple :

```
str += " (1870)";
```

Ici nous ajoutons un `constchar*` à un `QString` sans formalité. Pour convertir explicitement un `const char*` en un `QString`, il suffit d'utiliser une conversion `QString` ou encore d'appeler `fromAscii()` ou `fromLatin1()`. (Reportez-vous au Chapitre 17 pour obtenir une explication concernant la gestion des chaînes littérales et autres codages.)

Pour convertir un `QString` en un `const char *`, exécutez `toAscii()` ou `toLatin1()`. Ces fonctions retournent un `QByteArray`, qui peut être converti en un `const char*` en codant `QByteArray::data()` ou `QByteArray::constData()`. Par exemple :

```
printf("User: %s\n", str.toAscii().data());
```

Pour des raisons pratiques, Qt fournit la macro `qPrintable()` dont l'action est identique à celle de la séquence `toAscii().constData()` :

```
printf("User: %s\n", qPrintable(str));
```

Lorsque nous appelons `data()` ou `constData()` sur un `QByteArray`, la chaîne retournée appartient à l'objet `QByteArray`, ce qui signifie que nous n'avons pas à nous préoccuper de problèmes de pertes de mémoire. Qt se chargera de libérer la mémoire. D'autre part, nous devons veiller à ne pas utiliser le pointeur trop longtemps. Si le `QByteArray` n'est pas stocké dans une variable, il sera automatiquement supprimé à la fin de l'instruction.

La classe `QByteArray` possède une API très similaire à celle de `QString`. Des fonctions telles que `left()`, `right()`, `mid()`, `toLower()`, `toUpper()`, `trimmed()` et `simplified()` ont la même sémantique dans `QByteArray` que leurs homologues `QString`. `QByteArray` est utile pour stocker des données binaires brutes et des chaînes de texte codées en 8 bits. En général, nous conseillons d'utiliser `QString` plutôt que `QByteArray` pour stocker du texte, car cette classe supporte Unicode.

Pour des raisons de commodité, `QByteArray` s'assure automatiquement que l'octet suivant le dernier élément est toujours "0", ce qui facilite la transmission d'un `QByteArray` à une fonction recevant un `const char *`. `QByteArray` prend aussi en charge les caractères "0" intégrés, ce qui nous permet de l'utiliser pour stocker des données binaires arbitraires.

Dans certaines situations, il est nécessaire de stocker des données de types différents dans la même variable. Une approche consiste à coder les données en tant que `QByteArray` ou `QString`. Ces approches offrent une flexibilité totale, mais annule certains avantages du C++, et notamment la sécurité et l'efficacité des types. Qt offre une bien meilleure solution pour gérer des variables contenant différents types : `QVariant`.

La classe `QVariant` peut contenir des valeurs de nombreux types Qt, dont `QBrush`, `QColor`, `QCursor`, `QDateTime`, `QFont`, `QKeySequence`, `QPalette`, `QPen`, `QPixmap`, `QPoint`, `QRect`, `QRegion`, `QSize` et `QString`, ainsi que des types numériques C++ de base tels que `double` et `int`. Cette classe est également susceptible de contenir des conteneurs : `QMap<QString, QVariant>`, `QStringList` et `QList<QVariant>`.

Les variants sont abondamment utilisés par les classes d'affichage d'éléments, le module de base de données et `QSettings`, ce qui nous permet de lire et d'écrire des données d'élément, des données de base de données et des préférences utilisateur pour tout type compatible `QVariant`. Nous en avons déjà rencontré un exemple dans le Chapitre 3, où nous avons transmis un `QRect`, un `QStringList` et deux `bool` en tant que variants à `QSettings::setValue()`. Nous les avons récupérés ultérieurement comme variants.

Il est possible de créer arbitrairement des structures de données complexes utilisant `QVariant` en imbriquant des valeurs de types conteneur :

```
QMap<QString, QVariant> pearMap;
pearMap["Standard"] = 1.95;
pearMap["Organic"] = 2.25;

QMap<QString, QVariant> fruitMap;
fruitMap["Orange"] = 2.10;
fruitMap["Pineapple"] = 3.85;
fruitMap["Pear"] = pearMap;
```

Ici, nous avons créé un map avec des clés sous forme de chaînes (noms de produit) et des valeurs qui sont soit des nombres à virgule flottante (prix), soit des maps. Le map de niveau supérieur contient trois clés : "Orange", "Pear" et "Pineapple". La valeur associée à la clé "Pear" est un map qui contient deux clés ("Standard" et "Organic"). Lorsque nous parcourons un map contenant des variants, nous devons utiliser `type()` pour en contrôler le type de façon à pouvoir répondre de façon appropriée.

La création de telles structures de données peut sembler très séduisante, car nous pouvons ainsi organiser les données exactement comme nous le souhaitons. Mais le caractère pratique de `QVariant` est obtenu au détriment de l'efficacité et de la lisibilité. En règle générale, il convient de définir une classe C++ correcte pour stocker les données dès que possible.

`QVariant` est utilisé par le système métaobjet de Qt et fait donc partie du module *QtCore*. Néanmoins, lorsque nous le rattachons au module *QtGui*, `QVariant` peut stocker des types en liaison avec l'interface utilisateur graphique tels que `QColor`, `QFont`, `QIcon`, `QImage` et `QPixmap` :

```
QIcon icon("open.png");
QVariant variant = icon;
```

Pour récupérer la valeur d'un tel type à partir d'un `QVariant`, nous pouvons utiliser la fonction membre template `QVariant::Value<T>()` comme suit :

```
QIcon icon = variant.value<QIcon>();
```

La fonction `value<T>()` permet également d'effectuer des conversions entre des types de données non graphiques et `QVariant`, mais en pratique, nous utilisons habituellement les fonctions de conversion `to…()` (par exemple, `toString()`) pour les types non graphiques.

`QVariant` peut aussi être utilisé pour stocker des types de données personnalisés, en supposant qu'ils fournissent un constructeur par défaut et un constructeur de copie. Pour que ceci fonctionne, nous devons tout d'abord enregistrer le type au moyen de la macro `Q_DECLARE_METATYPE()`, généralement dans un fichier d'en-tête en dessous de la définition de classe :

```
Q_DECLARE_METATYPE(BusinessCard)
```

Cette technique nous permet d'écrire du code tel que celui-ci :

```
BusinessCard businessCard;
QVariant variant = QVariant::fromValue(businessCard);
...
if (variant.canConvert<BusinessCard>()) {
    BusinessCard card = variant.value<BusinessCard>();
    ...
}
```

Ces fonctions membre template ne sont pas disponibles pour MSVC 6, à cause d'une limite du compilateur. Si vous devez employer ce compilateur, utilisez plutôt les fonctions globales `qVariantFromValue()`, `qVariantValue<T>()`et `qVariantCanConvert<T>()`.

Si le type de données personnalisé possède des opérateurs << et >> pour effectuer des opérations de lecture et d'écriture dans un `QDataStream`, nous pouvons les enregistrer au moyen de `qRegisterMetaTypeStreamOperators<T>()`. Il est ainsi possible de stocker les préférences des types de données personnalisés à l'aide de `QSettings`, entre autres. Par exemple :

```
qRegisterMetaTypeStreamOperators<BusinessCard>("BusinessCard");
```

Dans ce chapitre, nous avons principalement étudié les conteneurs Qt, ainsi que `QString`, `QByteArray` et `QVariant`. En complément de ces classes, Qt fournit quelques autres conteneurs. `QPair<T1,T2>` en fait partie, qui stocke simplement deux valeurs et présente des similitudes avec `std::pair<T1,T2>`. `QBitArray` est un autre conteneur que nous utiliserons dans la première partie du Chapitre 19. Il existe enfin `QVarLengthArray<T,Prealloc>`, une alternative de bas niveau à `QVector<T>`. Comme il préalloue de la mémoire sur la pile et n'est pas implicitement partagé, sa surcharge est inférieure à celle de `QVector<T>`, ce qui le rend plus approprié pour les boucles étroites.

Les algorithmes de Qt, dont quelques-uns n'ayant pas été étudiés ici tels que `qCopyBackward()` et `qEqual()`, sont décrits dans la documentation de Qt à l'adresse **http://doc.trolltech.com/4.1/algorithms.html**. Vous trouverez des détails complémentaires concernant les conteneurs de Qt à l'adresse **http://doc.trolltech.com/4.1/containers.html**.

12

Entrées/Sorties

Au sommaire de ce chapitre

- ✔ Lire et écrire des données binaires
- ✔ Lire et écrire du texte
- ✔ Parcourir des répertoires
- ✔ Intégrer des ressources
- ✔ Communication inter-processus

Le besoin d'effectuer des opérations de lecture et d'écriture dans des fichiers ou sur un autre support est commun à presque toute application. Qt fournit un excellent support pour ces opérations par le biais de `QIODevice`, une abstraction puissante qui encapsule des "périphériques" capables de lire et d'écrire des blocs d'octets. Qt inclut les sous-classes `QIODevice` suivantes :

`QFile`	Accède aux fichiers d'un système de fichiers local et de ressources intégrées.
`QTemporaryFile`	Crée et accède à des fichiers temporaires du système de fichiers local.
`QBuffer`	Effectue des opérations de lecture et d'écriture de données dans un QByteArray.
`QProcess`	Exécute des programmes externes et gère la communication inter-processus.
`QTcpSocket`	Transfère un flux de données sur le réseau au moyen de TCP.
`QUdpSocket`	Envoie ou reçoit des datagrammes UDP sur le réseau.

`QProcess`, `QTcpSocket` et `QUdpSocket` sont des classes séquentielles, ce qui implique un accès unique aux données, en commençant par le premier octet et en progressant dans l'ordre jusqu'au dernier octet. `QFile`, `QTemporaryFile` et `QBuffer` sont des classes à accès aléatoire. Les octets peuvent donc être lus plusieurs fois à partir de tout emplacement. Elles fournissent la fonction `QIODevice::seek()` qui permet de repositionner le pointeur de fichier.

En plus de ces classes de périphérique, Qt fournit deux classes de flux de niveau plus élevé qui exécutent des opérations de lecture et écriture sur tout périphérique d'E/S : `QDataStream` pour les données binaires et `QTextStream` pour le texte. Ces classes gèrent des problèmes tels que le classement des octets et les codages de texte, de sorte que des applications Qt s'exécutant sur d'autres plates-formes ou pays puissent effectuer des opérations de lecture et d'écriture sur leurs fichiers respectifs. Ceci rend les classes d'E/S de Qt beaucoup plus pratiques que les classes C++ standard correspondantes, qui laissent la gestion de ces problèmes au programmeur de l'application.

`QFile` facilite l'accès aux fichiers individuels, qu'ils soient dans le système de fichier ou intégrés dans l'exécutable de l'application en tant que ressources. Pour les applications ayant besoin d'identifier des jeux complets de fichiers sur lesquels travailler, Qt fournit les classes `QDir` et `QFileInfo`, qui gèrent des répertoires et fournissent des informations concernant leurs fichiers.

La classe `QProcess` nous permet de lancer des programmes externes et de communiquer avec ceux-ci par le biais de leurs canaux d'entrée, de sortie et d'erreur standard (`cin`, `cout` et `cerr`). Nous pouvons définir les variables d'environnement et le répertoire de travail qui seront utilisés par l'application externe. Par défaut, la communication avec le processus est asynchrone (non bloquante), mais il est possible de parvenir à un blocage pour certaines opérations.

La gestion de réseau ainsi que la lecture et l'écriture XML sont des thèmes importants qui seront traités séparément dans leurs propres chapitres (Chapitre 14 et Chapitre 15).

Lire et écrire des données binaires

La façon la plus simple de charger et d'enregistrer des données binaires avec Qt consiste à instancier un `QFile`, à ouvrir le fichier et à y accéder par le biais d'un objet `QDataStream`. Ce dernier fournit un format de stockage indépendant de la plate-forme qui supporte les types C++ de base tels que `int` et `double`, de nombreux types de données Qt, dont `QByteArray`, `QFont`, `QImage`, `QPixmap`, `QString` et `QVariant` ainsi que des classes conteneur telles que `QList<T>` et `QMap<K,T>`.

Voici comment stocker un entier, un `QImage` et un `QMap<QString,QColor>` dans un fichier nommé `facts.dat` :

```
QImage image("philip.png");

QMap<QString, QColor> map;
map.insert("red", Qt::red);
```

```
map.insert("green", Qt::green);
map.insert("blue", Qt::blue);

QFile file("facts.dat");
if (!file.open(QIODevice::WriteOnly)) {
    cerr << "Cannot open file for writing: "
         << qPrintable(file.errorString()) << endl;
    return;
}

QDataStream out(&file);
out.setVersion(QDataStream::Qt_4_1);

out << quint32(0x12345678) << image << map;
```

Si nous ne pouvons pas ouvrir le fichier, nous en informons l'utilisateur et rendons le contrôle. La macro `qPrintable()` retourne un `constchar*` pour un `QString`. (Une autre approche consiste à exécuter `QString::toStdString()`, qui retourne un `std::string`, pour lequel `<iostream>` possède une surcharge `<<`.)

Si le fichier s'ouvre avec succès, nous créons un `qDataStream` et définissons son numéro de version. Le numéro de version est un entier qui influence la façon dont les types de données Qt sont représentés (les types de données C++ de base sont toujours représentés de la même façon). Dans Qt 4.1, le format le plus complet est la version 7. Nous pouvons soit coder en dur la constante `7`, soit utiliser le nom symbolique `QDataStream::Qt_4_1`.

Pour garantir que le nombre `0x12345678` sera bien enregistré en tant qu'entier non signé de 32 bits sur toutes les plates-formes, nous le convertissons en `quint32`, un type de données dont les 32 bits sont garantis. Pour assurer l'interopérabilité, `QDataStream` est basé par défaut sur Big-Endian, ce qui peut être modifié en appelant `setByteOrder()`.

Il est inutile de fermer explicitement le fichier, cette opération étant effectuée automatiquement lorsque la variable `QFile` sort de la portée. Si nous souhaitons vérifier que les données ont bien été écrites, nous appelons `flush()` et vérifions sa valeur de retour (`true` en cas de succès).

Le code destiné à lire les données reflète celui que nous avons utilisé pour les écrire :

```
quint32 n;
QImage image;
QMap<QString, QColor> map;

QFile file("facts.dat");
if (!file.open(QIODevice::ReadOnly)) {
    cerr << "Cannot open file for reading: "
         << qPrintable(file.errorString()) << endl;
    return;
}

QDataStream in(&file);
in.setVersion(QDataStream::Qt_4_1);

in >> n >> image >> map;
```

La version de `QDataStream` que nous employons pour la lecture est la même que celle utilisée pour l'écriture, ce qui doit toujours être le cas. En codant en dur le numéro de version, nous garantissons que l'application est toujours en mesure de lire et d'écrire les données.

`QDataStream` stocke les données de telle façon que nous puissions les lire parfaitement. Par exemple, un `QByteArray` est représenté sous la forme d'un décompte d'octets, suivi des octets eux-mêmes. `QDataStream` peut aussi être utilisé pour lire et écrire des octets bruts, sans entête de décompte d'octets, au moyen de `readRawBytes()` et de `writeRawBytes()`.

Lors de la lecture à partir d'un `QDataStream`, la gestion des erreurs est assez facile. Le flux possède une valeur `status()` qui peut être `QDataStream::Ok`, `QDataStream::ReadPastEnd` ou `QDataStream::ReadCorruptData`. Quand une erreur se produit, l'opérateur >> lit toujours zéro ou des valeurs vides. Nous pouvons ainsi lire un fichier entier sans nous préoccuper d'erreurs potentielles et vérifier la valeur de `status()` à la fin pour déterminer si ce que nous avons lu était valide.

`QDataStream` gère plusieurs types de données C++ et Qt. La liste complète est disponible à l'adresse **http://doc.trolltech.com/4.1/datastreamformat.html**. Nous pouvons également ajouter la prise en charge de nos propres types personnalisés en surchargeant les opérateurs << et >>. Voici la définition d'un type de données personnalisé susceptible d'être utilisé avec `QDataStream` :

```
class Painting
{
public:
   Painting() { myYear = 0; }
   Painting(const QString &title, const QString &artist, int year) {
      myTitle = title;
      myArtist = artist;
      myYear = year;
   }

   void setTitle(const QString &title) { myTitle = title; }
   QString title() const { return myTitle; }
   ...

private:
   QString myTitle;
   QString myArtist;
   int myYear;
};

QDataStream &operator<<(QDataStream &out, const Painting &painting);
QDataStream &operator>>(QDataStream &in, Painting &painting);
Voici comment nous implémenterions l'opérateur << :
QDataStream &operator<<(QDataStream &out, const Painting &painting)
{
   out << painting.title() << painting.artist()
       << quint32(painting.year());
   return out;
}
```

Pour émettre en sortie un `Painting`, nous émettons simplement deux `QString` et un `quint32`. A la fin de la fonction, nous retournons le flux. C'est une expression C++ courante qui nous permet d'utiliser une chaîne d'opérateurs `<<` avec un flux de sortie. Par exemple :

```
out << painting1 << painting2 << painting3;
```

L'implémentation de `operator>>()` est similaire à celle de `operator<<()` :

```
QDataStream &operator>>(QDataStream &in, Painting &painting)
{
   QString title;
   QString artist;
   quint32 year;

   in >> title >> artist >> year;
   painting = Painting(title, artist, year);
   return in;
}
```

Il existe plusieurs avantages à fournir des opérateurs de flux pour les types de données personnalisés. L'un d'eux est que nous pouvons ainsi transmettre des conteneurs qui utilisent le type personnalisé. Par exemple :

```
QList<Painting> paintings = ...;
out << paintings;
```

Nous pouvons lire les conteneurs tout aussi facilement :

```
QList<Painting> paintings;
in >> paintings;
```

Ceci provoquerait une erreur de compilateur si `Painting` ne supportait pas `<<` ou `>>`. Un autre avantage des opérateurs de flux pour les types personnalisés est que nous pouvons stocker les valeurs de ces types en tant que `QVariant`, ce qui les rend plus largement utilisables, par exemple par les `QSetting`. Ceci ne fonctionne que si nous enregistrons préalablement le type en exécutant `qRegisterMetaTypeStreamOperators<T>()`, comme expliqué dans le Chapitre 11.

Lorsque nous utilisons `QDataStream`, Qt se charge de lire et d'écrire chaque type, dont les conteneurs avec un nombre arbitraire d'éléments. Cette caractéristique nous évite de structurer ce que nous écrivons et d'appliquer une conversion à ce que nous lisons. Notre seule obligation consiste à nous assurer que nous lisons tous les types dans leur ordre d'écriture, en laissant à Qt le soin de gérer tous les détails.

`QDataStream` est utile à la fois pour nos formats de fichiers d'application personnalisés et pour les formats binaires standard. Nous pouvons lire et écrire des formats binaires standard en utilisant les opérateurs de flux sur les types de base (tels que `quint16` ou `float`) ou au moyen de `readRawBytes()` et de `writeRawBytes()`. Si le `QDataStream` est purement utilisé pour lire et écrire des types de données C++ de base, il est inutile d'appeler `setVersion()`.

Jusqu'à présent, nous avons chargé et enregistré les données avec la version codée en dur du flux sous la forme `QDataStream::Qt_4_1`. Cette approche est simple et sûre, mais elle présente un léger inconvénient : nous ne pouvons pas tirer parti des formats nouveaux ou mis à jour. Par exemple, si une version ultérieure de Qt ajoutait un nouvel attribut à `QFont` (en plus de sa famille, de sa taille, etc.) et que nous ayons codé en dur le numéro de version en `Qt_4_1`, cet attribut ne serait pas enregistré ou chargé. Deux solutions s'offrent à vous. La première approche consiste à intégrer le numéro de version `QDataStream` dans le fichier :

```
QDataStream out(&file);
out << quint32(MagicNumber) << quint16(out.version());
```

(`MagicNumber` est une constante qui identifie de façon unique le type de fichier.) Avec cette approche, nous écrivons toujours les données en utilisant la version la plus récente de `QDataStream`. Lorsque nous en venons à lire le fichier, nous lisons la version du flux :

```
quint32 magic;
quint16 streamVersion;

QDataStream in(&file);
in >> magic >> streamVersion;

if (magic != MagicNumber) {
    cerr << "File is not recognized by this application" << endl;
} else if (streamVersion > in.version()) {
    cerr << "File is from a more recent version of the application"
         << endl;
    return false;
}

in.setVersion(streamVersion);
```

Nous pouvons lire les données tant que la version du flux est inférieure ou égale à la version utilisée par l'application. Dans le cas contraire, nous signalons une erreur.

Si le format de fichier contient un numéro de version personnel, nous pouvons l'utiliser pour déduire le numéro de version du flux au lieu de le stocker explicitement. Supposons, par exemple, que le format de fichier est destiné à la version 1.3 de notre application. Nous pouvons alors écrire les données comme suit :

```
QDataStream out(&file);
out.setVersion(QDataStream::Qt_4_1);
out << quint32(MagicNumber) << quint16(0x0103);
```

Lorsque nous les relisons, nous déterminons quelle version de `QDataStream` utiliser selon le numéro de version de l'application :

```
QDataStream in(&file);
in >> magic >> appVersion;
```

```
if (magic != MagicNumber) {
    cerr << "File is not recognized by this application" << endl;
    return false;
} else if (appVersion > 0x0103) {
    cerr << "File is from a more recent version of the application"
         << endl;
    return false;
}

if (appVersion < 0x0103) {
   in.setVersion(QDataStream::Qt_3_0);
} else {
   in.setVersion(QDataStream::Qt_4_1);
}
```

Dans cet exemple, nous spécifions que tout fichier enregistré avec une version antérieure à la version 1.3 de l'application utilise la version de flux de données 4 (`Qt_3_0`) et que les fichiers enregistrés avec la version 1.3 de l'application utilisent la version de flux de données 7 (`Qt_4_1`).

En résumé, il existe trois stratégies pour gérer les versions de `QDataStream` : coder en dur le numéro de version, écrire et lire explicitement le numéro de version et utiliser différents numéros de version codés en dur en fonction de la version de l'application. Toutes peuvent être employées afin d'assurer que les données écrites par une ancienne version d'une application peuvent être lues par une nouvelle version. Une fois cette stratégie de gestion des versions de `QDataStream` choisie, la lecture et l'écriture de données binaires au moyen de Qt est à la fois simple et fiable.

Si nous souhaitons lire ou écrire un fichier en une seule fois, nous pouvons éviter l'utilisation de `QDataStream` et recourir à la place aux fonctions `write()` et `readAll()` de `QIODevice`. Par exemple :

```
bool copyFile(const QString &source, const QString &dest)
{
   QFile sourceFile(source);
   if (!sourceFile.open(QIODevice::ReadOnly))
       return false;

   QFile destFile(dest);
   if (!destFile.open(QIODevice::WriteOnly))
       return false;

   destFile.write(sourceFile.readAll());

   return sourceFile.error() == QFile::NoError
         && destFile.error() == QFile::NoError;
}
```

Sur la ligne de l'appel de `readAll()`, le contenu entier du fichier d'entrée est lu et placé dans un `QByteArray`. Il est alors transmis à la fonction `write()` pour être écrit dans le fichier de sortie. Le fait d'avoir toutes les données dans un `QByteArray` nécessite plus de mémoire que

de les lire élément par élément, mais offre quelques avantages. Nous pouvons exécuter, par exemple, `qCompress()` et `qUncompress()` pour les compresser et les décompresser.

Il existe d'autres scénarios où il est plus approprié d'accéder directement à `QIODevice` que d'utiliser `QDataStream`. `QIODevice` fournit une fonction `peek()` qui retourne les octets de donnée suivants sans changer l'emplacement du périphérique ainsi qu'une fonction `ungetChar()` qui permet de revenir un octet en arrière. Ceci fonctionne à la fois pour les périphériques d'accès aléatoire (tels que les fichiers) et pour les périphériques séquentiels (tels que les sockets réseau). Il existe également une fonction `seek()` qui définit la position des périphériques supportant l'accès aléatoire.

Les formats de fichier binaires offrent les moyens les plus souples et les plus compacts de stockage de données. `QDataStream` facilite l'accès aux données binaires. En plus des exemples de cette section, nous avons déjà étudié l'utilisation de `QDataStream` au Chapitre 4 pour lire et écrire des fichiers de tableur, et nous l'utiliserons de nouveau dans le Chapitre 19 pour lire et écrire des fichiers de curseur Windows.

Lire et écrire du texte

Les formats de fichiers binaires sont généralement plus compacts que ceux basés sur le texte, mais ils ne sont pas lisibles ou modifiables par l'homme. Si cela représente un problème, il est possible d'utiliser à la place les formats texte. Qt fournit la classe `QTextStream` pour lire et écrire des fichiers de texte brut et pour des fichiers utilisant d'autres formats texte, tels que HTML, XML et du code source. La gestion des fichiers XML est traitée dans le Chapitre 15.

`QTextStream` se charge de la conversion entre Unicode et le codage local du système ou tout autre codage, et gère de façon transparente les conventions de fin de ligne utilisées par les différents systèmes d'exploitation ("\r\n" sur Windows, "n" sur Unix et Mac OS X). `QTextStream` utilise le type `QChar` 16 bits comme unité de donnée fondamentale. En plus des caractères et des chaînes, `QTextStream` prend en charge les types numériques de base de C++, qu'il convertit en chaînes. Par exemple, le code suivant écrit "Thomas M. Disch : 334" dans le fichier `sf-book.txt` :

```
QFile file("sf-book.txt");
if (!file.open(QIODevice::WriteOnly)) {
    cerr << "Cannot open file for writing: "
         << qPrintable(file.errorString()) << endl;
    return;
}

QTextStream out(&file);
out << "Thomas M. Disch: " << 334 << endl;
```

L'écriture du texte est très facile, mais sa lecture peut représenter un véritable défit, car les données textuelles (contrairement aux données binaires écrites au moyen de `QDataStream`) sont fondamentalement ambiguës. Considérons l'exemple suivant :

```
out << "Norway" << "Sweden";
```

Si `out` est un `QTextStream`, les données véritablement écrites sont la chaîne "NorwaySweden". Nous ne pouvons pas vraiment nous attendre à ce que le code suivant lise les données correctement :

```
in >> str1 >> str2;
```

En fait, `str1` obtient le mot "NorwaySweden" entier et `str2` n'obtient rien. Ce problème ne se pose pas avec `QDataStream`, car la longueur de chaque chaîne est stockée devant les données.

Pour les formats de fichier complexes, un analyseur risque d'être requis. Un tel analyseur peut fonctionner en lisant les données caractère par caractère en utilisant un `>>` sur un `QChar`, ou ligne par ligne au moyen de `QTextStream::readLine()`. A la fin de cette section, nous présentons deux petits exemples. Le premier lit un fichier d'entrée ligne par ligne et l'autre le lit caractère par caractère. Pour ce qui est des analyseurs qui traitent le texte entier, nous pouvons lire le fichier complet en une seule fois à l'aide de `QTextStream::readAll()` si nous ne nous préoccupons pas de l'utilisation de la mémoire, ou si nous savons que le fichier est petit.

Par défaut, `QTextStream` utilise le codage local du système (par exemple, ISO 8859-1 ou ISO 8859-15 aux Etats-Unis et dans une grande partie de l'Europe) pour les opérations de lecture et d'écriture. Vous pouvez changer ceci en exécutant `setCodec()` comme suit :

```
stream.setCodec("UTF-8");
```

UTF-8 est un codage populaire compatible ASCII capable de représenter la totalité du jeu de caractères Unicode. Pour plus d'informations concernant Unicode et la prise en charge des codages de `QTextStream`, reportez-vous au Chapitre 17 (Internationalisation).

`QTextStream` possède plusieurs options modelées sur celles offertes par `<iostream>`. Elles peuvent être définies en transmettant des objets spéciaux, nommés *manipulateurs de flux*, au flux pour modifier son état. L'exemple suivant définit les options `showbase`, `uppercasedigits` et `hex` avant la sortie de l'entier 12345678, produisant le texte "0xBC614E" :

```
out << showbase << uppercasedigits << hex << 12345678;
```

Les options peuvent également être définies en utilisant les fonctions membres :

```
out.setNumberFlags(QTextStream::ShowBase
                   | QTextStream::UppercaseDigits);
out.setIntegerBase(16);
out << 12345678;
```

`setIntegerBase(int)`	
`0`	Détection automatique basée sur le préfixe (lors de la lecture)
`2`	Binaire
`8`	Octal
`10`	Décimal
`16`	Hexadécimal

`setNumberFlags(NumberFlags)`	
`ShowBase`	Affiche un préfixe si la base est 2 ("0b"), 8 ("0") ou 16 ("0x")
`ForceSign`	Affiche toujours le signe des nombres réels
`ForcePoint`	Place toujours le séparateur de décimale dans les nombres
`UppercaseBase`	Utilise les versions majuscules des préfixes de base ("0X", "0B")
`UppercaseDigits`	Utilise des lettres majuscules dans les nombres hexadécimaux

`setRealNumberNotation(RealNumberNotation)`	
`FixedNotation`	Notation à point fixe (par exemple, "0.000123")
`ScientificNotation`	Notation scientifique (par exemple, "1.234568e-04")
`SmartNotation`	Notation la plus compacte entre point fixe ou scientifique

`setRealNumberPrecision(int)`
Définit le nombre maximum de chiffres devant être générés (6 par défaut)

`setFieldWidth(int)`	
Définit la taille minimum d'un champ (0 par défaut)	
`setFieldAlignment(FieldAlignment)`	
`AlignLeft`	Force un alignement sur le côté gauche du champ
`AlignRight`	Force un alignement sur le côté droit du champ
`AlignCenter`	Force un alignement sur les deux côtés du champ
`AlignAccountingStyle`	Force un alignement entre le signe et le nombre

`setPadChar(QChar)`
Défini le caractère à utiliser pour l'alignement (espace par défaut)

Figure 12.1
Fonctions destinées à définir les options de `QTextStream`

Comme `QDataStream`, `QTextStream` agit sur une sous-classe de `QIODevice`, qui peut être un `QFile`, un `QTemporaryFile`, un `QBuffer`, un `QProcess`, un `QTcpSocket` ou un `QUdpSocket`. En outre, il pe**ut être utilisé directement sur un `QString`. Par exemple :**

```
QString str;
QTextStream(&str) << oct << 31 << " " << dec << 25 << endl;
```

Le contenu de `str` est donc "37 25/n", car le nombre décimal 31 est exprimé sous la forme 37 en base huit. Dans ce cas, il n'est pas nécessaire de définir un codage sur le flux, car `QString` est toujours Unicode.

Examinons un exemple simple de format de fichier basé sur du texte. Dans l'application Spreadsheet décrite dans la Partie 1, nous avons utilisé un format binaire pour stocker les données. Ces données consistaient en une séquence de triplets (*ligne, colonne, formule*), une pour chaque cellule non vide. Il n'est pas difficile d'écrire les données sous forme de texte. Voici un extrait d'une version révisée de `Spreadsheet::writeFile()`.

```
QTextStream out(&file);
for (int row = 0; row < RowCount; ++row) {
    for (int column = 0; column < ColumnCount; ++column) {
      QString str = formula(row, column);
      if (!str.isEmpty())
          out << row << " " << column << " " << str << endl;
    }
}
```

Nous avons utilisé un format simple, chaque ligne représentant une cellule avec des espaces entre la ligne et la colonne ainsi qu'entre la colonne et la formule. La formule contient des espaces, mais nous pouvons supposer qu'elle ne comporte aucun '/n' (qui insère un retour à la ligne). Examinons maintenant le code de lecture correspondant :

```
QTextStream in(&file);
while (!in.atEnd()) {
   QString line = in.readLine();
   QStringList fields = line.split(' ');
   if (fields.size() >= 3) {
      int row = fields.takeFirst().toInt();
      int column = fields.takeFirst().toInt();
      setFormula(row, column, fields.join(' '));
   }
}
```

Nous lisons les données de Spreadsheet ligne par ligne. La fonction `readLine()` supprime le '/n' de fin. `QString::split()` retourne une liste de chaînes de caractères qui est scindée à l'emplacement d'apparition du séparateur qui lui est fournit. Par exemple, la ligne "5 19 Total Value" résulte en une liste de quatre éléments ["5", "19", "Total", "Value"].

Si nous disposons au moins de trois champs, nous sommes prêts à extraire les données. La fonction `QStringList::takeFirst()` supprime le premier élément d'une liste et retourne l'élément supprimé. Nous l'utilisons pour extraire les nombres de ligne et de colonne.

Nous ne réalisons aucune vérification d'erreur. Si nous lisons une valeur de ligne ou de colonne non entière, `QString::toInt()` retourne `0`. Lorsque nous appelons `setFormula()`, nous devons concaténer les champs restants en une seule chaîne.

Dans notre deuxième exemple de `QTextStream`, nous utilisons une approche caractère par caractère pour implémenter un programme qui lit un fichier texte et émet en sortie le même texte avec les espaces de fin supprimés et toutes les tabulations remplacées par des espaces. Le programme accomplit cette tâche dans la fonction `tidyFile()` :

```
void tidyFile(QIODevice *inDevice, QIODevice *outDevice)
{
   QTextStream in(inDevice);
   QTextStream out(outDevice);

   const int TabSize = 8;
   int endlCount = 0;
   int spaceCount = 0;
   int column = 0;
   QChar ch;

   while (!in.atEnd()) {
      in >> ch;

      if (ch == '\n') {
         ++endlCount;
         spaceCount = 0;
         column = 0;
      } else if (ch == '\t') {
         int size = TabSize - (column % TabSize);
         spaceCount += size;
         column += size;
      } else if (ch == ' ') {
         ++spaceCount;
         ++column;
     } else {
         while (endlCount > 0) {
            out << endl;
            --endlCount;
            column = 0;
         }
         while (spaceCount > 0) {
            out << ' ';
            --spaceCount;
            ++column;
         }
         out << ch;
         ++column;
     }
   }
   out << endl;
}
```

Nous créons un `QTextStream` d'entrée et de sortie basé sur les `QIODevice` transmis à la fonction. Nous conservons trois éléments d'état : un décompte des nouvelles lignes, un décompte des espaces et l'emplacement de colonne courant dans la ligne en cours (pour convertir les tabulations en un nombre d'espaces correct).

L'analyse est faite dans une boucle `while` qui parcourt chaque caractère du fichier d'entrée. Le code présente quelques subtilités à certains endroits. Par exemple, bien que nous définissions `tabSize` en `8`, nous remplaçons les tabulations par le nombre d'espaces qui permet d'atteindre le taquet de tabulation suivant, au lieu de remplacer grossièrement chaque tabulation par huit espaces. Dans le cas d'une nouvelle ligne, d'une nouvelle tabulation ou d'un nouvel espace, nous mettons simplement à jour les données d'état. Pour les autres types de caractères, nous produisons une sortie. Avant d'écrire le caractère nous introduisons tous les espaces et les nouvelles lignes nécessaires (pour respecter les lignes vierges et préserver le retrait) et mettons à jour l'état.

```
int main()
{
   QFile inFile;
   QFile outFile;

   inFile.open(stdin, QFile::ReadOnly);
   outFile.open(stdout, QFile::WriteOnly);

   tidyFile(&inFile, &outFile);

   return 0;
}
```

Pour cet exemple, nous n'avons pas besoin d'objet `QApplication`, car nous n'utilisons que les classes d'outils de Qt. Vous trouverez la liste de toutes les classes d'outils à l'adresse **http://doc.trolltech.com/4.1/tools.html**. Nous avons supposé que le programme est utilisé en tant que filtre, par exemple :

```
tidy < cool.cpp > cooler.cpp
```

Il serait facile de le développer afin de gérer les noms de fichier qui seraient transmis sur la ligne de commande ainsi que pour filtrer `cin` en `cout`.

Comme il s'agit d'une application de console, le fichier `.pro` diffère légèrement de ceux que nous avons rencontrés pour les applications GUI :

```
TEMPLATE     = app
QT           = core
CONFIG      += console
CONFIG      -= app_bundle
SOURCES      = tidy.cpp
```

Nous n'établissons de liaison qu'avec `QtCore`, car nous n'utilisons aucune fonctionnalité GUI. Puis nous spécifions que nous souhaitons activer la sortie de la console sur Windows et que nous ne voulons pas que l'application soit hébergée dans un package sur Mac OS X.

Pour lire et écrire des fichiers ASCII ou ISO 8859-1 (Latin-1) bruts, il est possible d'utiliser directement l'API de `QIODevice` au lieu de `QTextStream`. Mais cette méthode est rarement conseillée dans la mesure où la plupart des applications doivent prendre en charge d'autres codages à un stade ou à un autre, et où seul `QTextStream` offre une prise en charge parfaite de ces codages. Si vous souhaitez néanmoins écrire le texte directement dans un `QIODevice`, vous devez spécifier explicitement la balise `QIODevice::Text` dans la fonction `open()`. Par exemple :

```
file.open(QIODevice::WriteOnly | QIODevice::Text);
```

Lors de l'écriture, cette balise indique à `QIODevice` de convertir les caractères '\n' en des séquences "\r\n" sur Windows. Lors de la lecture, elle indique au périphérique d'ignorer les caractères 'r' sur toutes les plates-formes. Nous pouvons alors supposer que la fin de chaque ligne est marquée par un caractère de nouvelle ligne 'n' quelle que soit la convention de fin de ligne utilisée par le système d'exploitation.

Parcourir les répertoires

La classe `QDir` fournit un moyen indépendant de la plate-forme de parcourir les répertoires et de récupérer des informations concernant les fichiers. Pour déterminer comment `QDir` est utilisée, nous allons écrire une petite application de console qui calcule l'espace occupé par toutes les images d'un répertoire particulier et de tous ses sous-répertoires, quelle que soit leur profondeur.

Le cœur de l'application est la fonction `imageSpace()`, qui calcule récursivement la taille cumulée des images d'un répertoire donné :

```
qlonglong imageSpace(const QString &path)
{
   QDir dir(path);
   qlonglong size = 0;

   QStringList filters;
   foreach (QByteArray format, QImageReader::supportedImageFormats())
      filters += "*." + format;

   foreach (QString file, dir.entryList(filters, QDir::Files))
      size += QFileInfo(dir, file).size();

   foreach (QString subDir, dir.entryList(QDir::Dirs
                                          | QDir::NoDotAndDotDot))
      size += imageSpace(path + QDir::separator() + subDir);

   return size;
}
```

Nous commençons par créer un objet `QDir` en utilisant un chemin donné, qui peut être relatif au répertoire courant ou absolu. Nous transmettons deux arguments à la fonction `entryList()`. Le premier est une liste de filtres de noms de fichiers. Ils peuvent contenir les caractères génériques '*' et '?.' Dans cet exemple, nous réalisons un filtrage de façon à n'inclure que les formats de fichiers susceptibles d'être lus par `QImage`. Le second argument spécifie le type d'entrée souhaité (fichiers normaux, répertoires, etc.).

Nous parcourons la liste de fichiers, en additionnant leurs tailles. La classe `QFileInfo` nous permet d'accéder aux attributs d'un fichier, tels que la taille, les autorisations, le propriétaire et l'horodateur de ce fichier.

Le deuxième appel de `entryList()` récupère tous les sous-répertoires de ce répertoire. Nous les parcourons et nous appelons `imageSpace()` récursivement pour établir la taille cumulée de leurs images.

Pour créer le chemin de chaque sous-répertoire, nous combinons de chemin du répertoire en cours avec le nom du sous-répertoire, en les séparant par un slash (/).

`QDir` traite '/' comme un séparateur de répertoires sur toutes les plates-formes. Pour présenter les chemins à l'utilisateur, nous pouvons appeler la fonction statique `QDir::convertSeparators()` qui convertit les slash en séparateurs spécifiques à la plate-forme.

Ajoutons une fonction `main()` à notre petit programme :

```
int main(int argc, char *argv[])
{
   QCoreApplication app(argc, argv);
   QStringList args = app.arguments();

   QString path = QDir::currentPath();
   if (args.count() > 1)
      path = args[1];

   cout << "Space used by images in " << qPrintable(path)
          << " and its subdirectories is " << (imageSpace(path) / 1024)
          << " KB" << endl;

   return 0;
}
```

Nous utilisons `QDir::currentPath()` pour initialiser le chemin vers le répertoire courant. Nous aurions aussi pu faire appel à `QDir::homePath()` pour l'initialiser avec le répertoire de base de l'utilisateur. Si ce dernier a spécifié un chemin sur la ligne de commande, nous l'utilisons à la place. Nous appelons enfin notre fonction `imageSpace()` pour calculer l'espace occupé par les images.

La classe `QDir` fournit d'autres fonctions liées aux répertoires et aux fichiers, dont `entryInfoList()` (qui retourne une liste d'objets `QFileInfo`), `rename()`, `exists()`, `mkdir()` et `rmdir()`. La classe `QFile` fournit des fonctions utilitaires statiques, dont `remove()` et `exists()`.

Intégration des ressources

Jusqu'à présent, nous avons étudié l'accès aux données dans des périphériques externes, mais avec Qt, il est également possible d'intégrer du texte ou des données binaires dans l'exécutable de l'application. Pour ce faire, il convient d'utiliser le système de ressources de Qt. Dans les autres chapitres, nous avons utilisé les fichiers de ressources pour intégrer des images dans l'exécutable, mais il est possible d'intégrer tout type de fichier. Les fichiers intégrés peuvent être lus au moyen de `QFile`, exactement comme les fichiers normaux d'un système de fichiers.

Les ressources sont converties en code C++ par `rcc`, le compilateur de ressources de Qt. Nous pouvons demander à `qmake` d'inclure des règles spéciales d'exécution de `rcc` en ajoutant cette ligne au fichier `.pro` :

```
RESOURCES = myresourcefile.qrc
```

`myresourcefile.qrc` est un fichier XML qui répertorie les fichiers à intégrer dans l'exécutable.

Imaginons que nous écrivions une application destinée à répertorier des coordonnées. Pour des raisons pratiques, nous souhaitons intégrer les indicatifs téléphoniques internationaux dans l'exécutable. Si le fichier se situe dans le répertoire `datafiles` de l'application, le fichier de ressources serait le suivant :

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
   <file>datafiles/phone-codes.dat</file>
</qresource>
</RCC>
```

Depuis l'application, les ressources sont identifiées par le préfixe de chemin `:/`. Dans cet exemple, le chemin du fichier contenant les indicatifs téléphoniques est `:/datafiles/phone-codes.dat` et peut être lu comme tout autre fichier en utilisant `QFile`.

L'intégration de données dans l'exécutable présente plusieurs avantages : les données ne peuvent être perdues et cette opération permet la création d'exécutables véritablement autonomes (si une liaison statique est également utilisée). Les inconvénients sont les suivants : si les données intégrées doivent être changées, il est impératif de remplacer l'exécutable entier, et la taille de ce dernier sera plus importante car il doit s'adapter aux données intégrées.

Le système de ressources de Qt fournit des fonctionnalités supplémentaires, telles que la prise en charge des alias de noms de fichiers et la localisation. Ces fonctionnalités sont documentées à l'adresse **http://doc.trolltech.com/4.1/resources.html**.

Communication inter-processus

La classe `QProcess` nous permet d'exécuter des programmes externes et d'interagir avec ceux-ci. La classe fonctionne de façon asynchrone, effectuant sa tâche à l'arrière-plan de sorte que l'interface utilisateur reste réactive. `QProcess` émet des signaux pour nous indiquer quand le processus externe détient des données ou a terminé.

Nous allons réviser le code d'une petite application qui fournit une interface utilisateur pour un programme externe de conversion des images. Pour cet exemple, nous nous reposons sur le programme `ImageMagick convert`, qui est disponible gratuitement sur toutes les plates-formes principales. (Voir Figure 12.2)

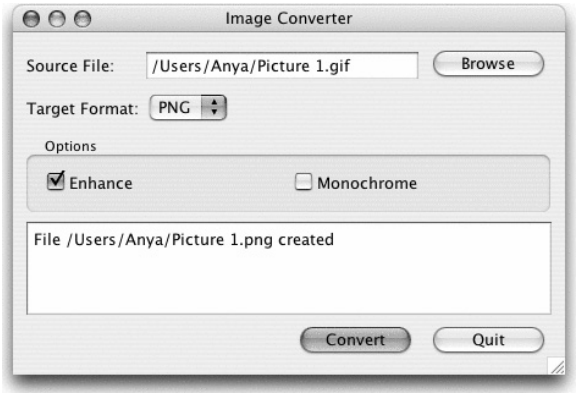


Figure 12.2
L'application Image Converter

L'interface utilisateur a été créée dans *Qt Designer*. Le fichier `.ui` se trouve sur le site web de Pearson, www.pearson.fr, à la page dédiée à ce livre. Ici, nous allons nous concentrer sur la sous-classe qui hérite de la classe `Ui::ConvertDialog` générée par le compilateur d'interface utilisateur. Commençons par l'en-tête :

```
#ifndef CONVERTDIALOG_H
#define CONVERTDIALOG_H

#include <QDialog>
#include <QProcess>

#include "ui_convertdialog.h"

class ConvertDialog : public QDialog, public Ui::ConvertDialog
{

   Q_OBJECT

public:
   ConvertDialog(QWidget *parent = 0);
```

```
private slots:
   void on_browseButton_clicked();
   void on_convertButton_clicked();
   void updateOutputTextEdit();
   void processFinished(int exitCode, QProcess::ExitStatus exitStatus);
   void processError(QProcess::ProcessError error);

private:
   QProcess process;
   QString targetFile;
};

#endif
```

L'en-tête est conforme à celui d'une sous-classe de formulaire *Qt Designer*. Grâce au mécanisme de connexion automatique de *Qt Designer*, les slots `on_browseButton_clicked()` et `on_convertButton_clicked()` sont automatiquement connectés aux signaux `clicked()` des boutons `Browse` et `Convert`.

```
ConvertDialog::ConvertDialog(QWidget *parent)
  : QDialog(parent)
{
   setupUi(this);

connect(&process, SIGNAL(readyReadStandardError()),
        this, SLOT(updateOutputTextEdit()));
connect(&process, SIGNAL(finished(int, QProcess::ExitStatus)),
        this, SLOT(processFinished(int, QProcess::ExitStatus)));
connect(&process, SIGNAL(error(QProcess::ProcessError)),
        this, SLOT(processError(QProcess::ProcessError)));
}
```

L'appel de `setupUi()` crée et positionne tous les widgets du formulaire, établit les connexions signal/slot pour les slots `on_objectName_signalName()` et connecte le bouton `Quit` à `QDialog::accept()`. Nous connectons ensuite manuellement trois signaux de l'objet `QProcess` à trois slots privés. A chaque fois que le processus externe va détecter des données sur son `cerr`, il les gérera avec `updateOutputTextEdit()`.

```
void ConvertDialog::on_browseButton_clicked()
{
   QString initialName = sourceFileEdit->text();
   if (initialName.isEmpty())
       initialName = QDir::homePath();
   QString fileName =
               QFileDialog::getOpenFileName(this, tr("Choose File"),
                                            initialName);
   fileName = QDir::convertSeparators(fileName);
   if (!fileName.isEmpty()) {
      sourceFileEdit->setText(fileName);
      convertButton->setEnabled(true);
   }
}
```

Le signal `clicked()` du bouton `Browse` est automatiquement connecté au slot `on_browseButton_clicked()` par `setupUi()`. Si l'utilisateur a préalablement sélectionné un fichier, nous initialisons la boîte de dialogue avec le nom de ce fichier. Sinon, nous prenons le répertoire de base de l'utilisateur.

```
void ConvertDialog::on_convertButton_clicked()
{
   QString sourceFile = sourceFileEdit->text();
   targetFile = QFileInfo(sourceFile).path() + QDir::separator()
                + QFileInfo(sourceFile).baseName() + "."
                + targetFormatComboBox->currentText().toLower();
   convertButton->setEnabled(false);
   outputTextEdit->clear();

   QStringList args;
   if (enhanceCheckBox->isChecked())
       args << "-enhance";
   if (monochromeCheckBox->isChecked())
       args << "-monochrome";
   args << sourceFile << targetFile;

   process.start("convert", args);
}
```

Lorsque l'utilisateur clique sur le bouton `Convert`, nous copions le nom du fichier source et changeons l'extension afin qu'elle corresponde au format du fichier cible.

Nous utilisons le séparateur de répertoires spécifique à la plate-forme ('/' ou '\', disponible sous la forme `QDir::separator()`) au lieu de coder les slash en dur, car le nom du fichier sera visible pour l'utilisateur.

Nous désactivons alors le bouton `Convert` pour éviter que l'utilisateur ne lance accidentellement plusieurs conversions, et nous effaçons l'éditeur de texte qui nous permet d'afficher les informations d'état.

Pour lancer le processus externe, nous appelons `QProcess::start()` avec le nom du programme à exécuter (`convert`) et tous les arguments requis. Dans ce cas, nous transmettons les balises `–enhance` et `–monochrome` si l'utilisateur a coché les options appropriées, suivies des noms des fichiers source et cible. Le programme `convert` déduit la conversion requise à partir des extensions de fichier.

```
void ConvertDialog::updateOutputTextEdit()
{
   QByteArray newData = process.readAllStandardError();
   QString text = outputTextEdit->toPlainText()
                  + QString::fromLocal8Bit(newData);
   outputTextEdit->setPlainText(text);
}
```

Lorsque le processus externe effectue une opération d'écriture dans le `cerr`, le slot `updateOutputTextEdit()` est appelé. Nous lisons le texte d'erreur et l'ajoutons au texte existant de `QTextEdit`.

```
void ConvertDialog::processFinished(int exitCode,
                                    QProcess::ExitStatus exitStatus)
{
   if (exitStatus == QProcess::CrashExit) {
       outputTextEdit->append(tr("Conversion program crashed"));
   } else if (exitCode != 0) {
       outputTextEdit->append(tr("Conversion failed"));
   } else {
       outputTextEdit->append(tr("File %1 created").arg(targetFile));
   }
   convertButton->setEnabled(true);
}
```

Une fois le processus terminé, nous faisons connaître le résultat à l'utilisateur et activons le bouton `Convert`.

```
void ConvertDialog::processError(QProcess::ProcessError error)
{
   if (error == QProcess::FailedToStart) {
       outputTextEdit->append(tr("Conversion program not found"));
       convertButton->setEnabled(true);
   }
}
```

Si le processus ne peut être lancé, `QProcess` émet `error()` au lieu de `finished()`. Nous rapportons toute erreur et activons le bouton `Click`.

Dans cet exemple, nous avons effectué les conversions de fichier de façon asynchrone – nous avons demandé à `QProcess` d'exécuter le programme `convert` et de rendre immédiatement le contrôle à l'application. Cette méthode laisse l'interface utilisateur réactive puisque le processus s'exécute à l'arrière-plan. Mais dans certains cas, il est nécessaire que le processus externe soit terminé avant de pouvoir poursuivre avec notre application. Dans de telles situations, `QProcess` doit agir de façon synchrone.

Les applications qui prennent en charge l'édition de texte brut dans l'éditeur de texte préféré de l'utilisateur sont typiques de celles dont le comportement doit être synchrone. L'implémentation s'obtient facilement en utilisant `QProcess`. Supposons, par exemple, que le texte brut se trouve dans un `QTextEdit`, et que vous fournissiez un bouton `Edit` sur lequel l'utilisateur peut cliquer, connecté à un slot `edit()`.

```
void ExternalEditor::edit()
{
   QTemporaryFile outFile;
   if (!outFile.open())
       return;

   QString fileName = outFile.fileName();
```

```
    QTextStream out(&outFile);
    out << textEdit->toPlainText();
    outFile.close();

    QProcess::execute(editor, QStringList() << options << fileName);

    QFile inFile(fileName);
    if (!inFile.open(QIODevice::ReadOnly))
        return;

    QTextStream in(&inFile);
    textEdit->setPlainText(in.readAll());
}
```

Nous utilisons `QTemporaryFile` pour créer un fichier vide avec un nom unique. Nous ne spécifions aucun argument pour `QTemporaryFile::open()` puisqu'elle est judicieusement définie pour ouvrir en mode écriture/lecture par défaut. Nous écrivons le contenu de l'éditeur de texte dans le fichier temporaire, puis nous fermons ce dernier car certains éditeurs de texte ne peuvent travailler avec des fichiers déjà ouverts.

La fonction statique `QProcess::execute()` exécute un processus externe et provoque un blocage jusqu'à ce que le processus soit terminé. L'argument `editor` est un `QString` contenant le nom d'un exécutable éditeur ("gvim", par exemple). L'argument `options` est un `QStringList` (contenant un élément, "-f", si nous utilisons `gvim`).

Une fois que l'utilisateur a fermé l'éditeur de texte, le processus se termine ainsi que l'appel de `execute()`. Nous ouvrons alors le fichier temporaire et lisons son contenu dans le `QTextEdit`. `QTemporaryFile` supprime automatiquement le fichier temporaire lorsque l'objet sort de la portée.

Les connexions signal/slot ne sont pas nécessaires lorsque `QProcess` est utilisé de façon synchrone. Si un contrôle plus fin que celui fourni par la fonction statique `execute()` est requis, nous pouvons utiliser une autre approche qui implique de créer un objet `QProcess` et d'appeler `start()` sur celui-ci, puis de forcer un blocage en appelant `QProcess::waitForStarted()`, et en cas de succès, en appelant `QProcess::waitForFinished()`. Reportez-vous à la documentation de référence de `QProcess` pour trouver un exemple utilisant cette approche.

Dans cette section, nous avons exploité `QProcess` pour obtenir l'accès à une fonctionnalité préexistante. L'utilisation d'applications déjà existantes représente un gain de temps et vous évite d'avoir à résoudre des problèmes très éloignés de l'objectif principal de votre application. Une autre façon d'accéder à une fonctionnalité préexistante consiste à établir une liaison vers une bibliothèque qui la fournit. Si une telle bibliothèque n'existe pas, vous pouvez aussi envisager d'encapsuler votre application de console dans un `QProcess`.

`QProcess` peut également servir à lancer d'autres applications GUI, telles qu'un navigateur Web ou un client email. Si, cependant, votre objectif est la communication entre applications et non la simple exécution de l'une à partir de l'autre, il serait préférable de les faire communiquer directement en utilisant les classes de gestion de réseau de Qt ou l'extension ActiveQt sur Windows.

13

Les bases de données

Au sommaire de ce chapitre

- ✔ Connexion et exécution de requêtes
- ✔ Présenter les données sous une forme tabulaire
- ✔ Implémenter des formulaires maître/détail

Le module *QtSql* fournit une interface indépendante de la plate-forme pour accéder aux bases de données. Cette interface est prise en charge par un ensemble de classes qui utilisent l'architecture modèle/vue de Qt pour intégrer la base de données à l'interface utilisateur. Ce chapitre suppose une certaine familiarité avec les classes modèle/vue de Qt, traitées dans le Chapitre 10.

Une connexion de base de données est représentée par un objet `QSqlDatabase`. Qt utilise des pilotes pour communiquer avec les diverses API de base de données. Qt Desktop Edition inclut les pilotes suivants :

Pilote	*Base de données*
QDB2	IBM DB2 version 7.1 et ultérieure
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	Oracle (Oracle Call Interface)
QODBC	ODBC (inclut Microsoft SQL Server)
QPSQL	PostgreSQL versions 6.x et 7.x
QSQLITE	SQLite version 3 et ultérieure
QSQLITE2	SQLite version 2
QTDS	Sybase Adaptive Server

Tous les pilotes ne sont pas fournis avec Qt Open Source Edition, en raison de restrictions de licence. Lors de la configuration de Qt, nous pouvons choisir entre inclure directement les pilotes SQL à Qt et les créer en tant que plugin. Qt est fourni avec la base de données SQLite, une base de données in-process (qui s'intègre aux applications) de domaine public.

Pour les utilisateurs familiarisés avec la syntaxe SQL, la classe `QSqlQuery` représente un bon moyen d'exécuter directement des instructions SQL arbitraires et de gérer leurs résultats. Pour les utilisateurs qui préférent une interface de base de données plus évoluée masquant la syntaxe SQL, `QSqlTableModel` et `QSqlRelationalTableModel` fournissent des abstractions acceptables. Ces classes représentent une table SQL de la même façon que les autres classes modèle de Qt (traitées dans le Chapitre 10).Elles peuvent être utilisées de façon autonome pour parcourir et modifier des données dans le code, ou encore être associées à des vues par le biais desquelles les utilisateurs finaux peuvent afficher et modifier les données.

Connexion et exécution de requêtes

Pour exécuter des requêtes SQL, nous devons tout d'abord établir une connexion avec une base de données. En règle générale, les connexions aux bases de données sont définies dans une fonction séparée que nous appelons au lancement de l'application. Par exemple :

```
bool createConnection()
{
   QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
```

```
    db.setHostName("mozart.konkordia.edu");
    db.setDatabaseName("musicdb");
    db.setUserName("gbatstone");
    db.setPassword("T17aV44");
    if (!db.open()) {
       QMessageBox::critical(0, QObject::tr("Database Error"),
                                          db.lastError().text());
       return false;
    }
    return true;
}
```

Dans un premier temps, nous appelons `QSqlDatabase::addDatabase()` pour créer un objet `QSqlDatabase`. Le premier argument de `addDatabase()` spécifie quel pilote doit être utilisé par Qt pour accéder à la base de données. Dans ce cas, nous utilisons MySQL.

Nous définissons ensuite le nom de l'hôte de la base de données, le nom de cette base de données, le nom d'utilisateur et le mot de passe, puis nous ouvrons la connexion. Si `open()` échoue, nous affichons un message d'erreur.

Nous appelons généralement `createConnection()` dans `main()` :

```
int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   if (!createConnection())
       return 1;
   ...
   return app.exec();
}
```

Une fois qu'une connexion est établie, nous pouvons utiliser `QSqlQuery` pour exécuter toute instruction SQL supportée par la base de données concernée. Par exemple, voici comment exécuter une instruction `SELECT` :

```
QSqlQuery query;
query.exec("SELECT title, year FROM cd WHERE year >= 1998");
```

Après l'appel d'`exec()`, nous pouvons parcourir l'ensemble de résultats de la requête :

```
while (query.next()) {
   QString title = query.value(0).toString();
   int year = query.value(1).toInt();
   cerr << qPrintable(title) << ": " << year << endl;
}
```

Au premier appel de `next()`, `QSqlQuery` est positionné sur le *premier* enregistrement de l'ensemble de résultats. Les appels ultérieurs à `next()` permettent d'avancer le pointeur vers les enregistrements successifs, jusqu'à ce que la fin soit atteinte. A ce stade, `next()` retourne `false`. Si l'ensemble de résultats est vide (ou si la requête a échoué), le premier appel à `next()` retourne `false`.

La fonction `value()` retourne la valeur d'un champ en tant que `QVariant`. Les champs sont numérotés en partant de 0 dans l'ordre fourni dans l'instruction `SELECT`. La classe `QVariant` peut contenir de nombreux types Qt et C++, dont `int` et `QString`. Les différents types de données susceptibles d'être stockés dans une base de données sont transformés en types Qt et C++ correspondants et stockés en tant que `QVariant`. Par exemple, un `VARCHAR` est représenté en tant que `QString` et un `DATETIME` en tant que `QDateTime`.

`QSqlQuery` fournit quelques autres fonctions destinées à parcourir l'ensemble de résultats : `first()`, `last()`, `previous()` et `seek()`. Ces fonctions sont pratiques, mais pour certaines bases de données elles peuvent s'avérer plus lentes et plus gourmandes en mémoire que `next()`. Pour optimiser facilement dans le cas de jeux de données de taille importante, nous pouvons appeler `QSqlQuery::setForwardOnly(true)` avant d'appeler `exec()`, puis seulement utiliser `next()` pour parcourir l'ensemble de résultats.

Nous avons précédemment présenté la requête SQL comme un argument de `QSqlQuery::exec()`, mais il est également possible de la transmettre directement au constructeur, qui l'exécute immédiatement :

```
QSqlQuery query("SELECT title, year FROM cd WHERE year >= 1998");
```

Nous pouvons contrôler l'existence d'une erreur en appelant `isActive()` sur la requête :

```
if (!query.isActive())
   QMessageBox::warning(this, tr("Database Error"),
                        query.lastError().text());
```

Si aucune erreur n'apparaît, la requête devient "active" et nous pouvons utiliser `next()` pour parcourir l'ensemble de résultats.

Il est presque aussi facile de réaliser un `INSERT` que d'effectuer un `SELECT` :

```
QSqlQuery query("INSERT INTO cd (id, artistid, title, year) "
                "VALUES (203, 102, 'Living in America', 2002)");
```

Après cette opération, `numRowsAffected()` retourne le nombre de lignes affectées par l'instruction SQL (ou -1 en cas d'erreur).

Si nous devons insérer de nombreux enregistrements, ou si nous souhaitons éviter la conversion de valeurs en chaînes, nous pouvons recourir à `prepare()` pour exécuter une requête contenant des emplacements réservés puis lier les valeurs à insérer. Qt prend en charge à la fois la syntaxe de style ODBC et celle de style Oracle pour les espaces réservés, en utilisant le support natif lorsqu'il est disponible et en le simulant dans les autres cas. Voici un exemple utilisant la syntaxe de style Oracle avec des espaces réservés nommés :

```
QSqlQuery query;
query.prepare("INSERT INTO cd (id, artistid, title, year) "
              "VALUES (:id, :artistid, :title, :year)");
query.bindValue(":id", 203);
query.bindValue(":artistid", 102);
query.bindValue(":title", "Living in America");
```

```
query.bindValue(":year", 2002);
query.exec();
```

Voici le même exemple utilisant des espaces réservés positionnels de style ODBC :

```
QSqlQuery query;
query.prepare("INSERT INTO cd (id, artistid, title, year) "
              "VALUES (?, ?, ?, ?)");
query.addBindValue(203);
query.addBindValue(102);
query.addBindValue("Living in America");
query.addBindValue(2002);
query.exec();
```

Après l'appel à `exec()`, nous pouvons appeler `bindValue()` ou `addBindValue()` pour lier de nouvelles valeurs, puis nous appelons de nouveau `exec()` pour exécuter la requête avec ces nouvelles valeurs.

Les espaces réservés sont souvent utilisés pour des données binaires contenant des caractères non ASCII ou n'appartenant pas au jeu de caractères Latin-1. A l'arrière-plan, Qt utilise Unicode avec les bases de données qui prennent en charge cette norme. Pour les autres, Qt convertit de façon transparente les chaînes en codage approprié.

Qt supporte les transactions SQL sur les bases de données pour lesquelles elles sont disponibles. Pour lancer une transaction, nous appelons `transaction()` sur l'objet `QSqlDatabase` qui représente la connexion de base de données. Pour mettre fin à la transaction, nous appelons soit `commit()`, soit `rollback()`. Voici, par exemple, comment rechercher une clé étrangère et exécuter une instruction `INSERT` dans une transaction :

```
QSqlDatabase::database().transaction();
QSqlQuery query;
query.exec("SELECT id FROM artist WHERE name = 'Gluecifer'");
if (query.next()) {
   int artistId = query.value(0).toInt();
   query.exec("INSERT INTO cd (id, artistid, title, year) "
                      "VALUES (201, " + QString::number(artistId)
                    + ", 'Riding the Tiger', 1997)");
}
QSqlDatabase::database().commit();
```

La fonction `QSqlDatabase::database()` retourne un objet `QSqlDatabase` représentant la connexion créée dans `createConnection()`. Si une transaction ne peut être démarrée, `QSqlDatabase::transaction()` retourne `false`. Certaines bases de données ne supportent pas les transactions. Dans cette situation, les fonctions `transaction()`, `commit()` et `rollback()` n'ont aucune action. Nous pouvons déterminer si une base de données prend en charge les transactions en exécutant `hasFeature()` sur le `QSqlDriver` associé à cette base :

```
QSqlDriver *driver = QSqlDatabase::database().driver();
if (driver->hasFeature(QSqlDriver::Transactions))
   ...
```

Plusieurs autres fonctionnalités de bases de données peuvent être testées, notamment la prise en charge des BLOB (objets binaires volumineux), d'Unicode et des requêtes préparées.

Dans les exemples fournis jusqu'à présent, nous avons supposé que l'application utilise une seule connexion de base de données. Pour créer plusieurs connexions, il est possible de transmettre un nom en tant que second argument à `addDatabase()`. Par exemple :

```
QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL", "OTHER");
db.setHostName("saturn.mcmanamy.edu");
db.setDatabaseName("starsdb");
db.setUserName("hilbert");
db.setPassword("ixtapa7");
```

Nous pouvons alors obtenir un pointeur vers l'objet `QSqlDatabase` en transmettant le nom à `QSqlDatabase::database()` :

```
QSqlDatabase db = QSqlDatabase::database("OTHER");
```

Pour exécuter des requêtes en utilisant l'autre connexion, nous transmettons l'objet `QSqlDatabase` au constructeur de `QSqlQuery` :

```
QSqlQuery query(db);
query.exec("SELECT id FROM artist WHERE name = 'Mando Diao'");
```

Des connexions multiples peuvent s'avérer utiles si vous souhaitez effectuer plusieurs transactions à la fois, chaque connexion ne pouvant gérer qu'une seule transaction. Lorsque nous utilisons les connexions de base de données multiples, nous pouvons toujours avoir une connexion non nommée qui sera exploitée par `QSqlQuery` si aucune connexion n'est spécifiée.

En plus de `QSqlQuery`, Qt fournit la classe `QSqlTableModel` comme interface de haut niveau, ce qui permet d'éviter l'emploi du code SQL brut pour réaliser les opérations SQL les plus courantes (`SELECT`, `INSERT`, `UPDATE` et `DELETE`). La classe peut être utilisée de façon autonome pour manipuler une base de données sans aucune implication GUI. Elle peut également être utilisée comme source de données pour `QListView` ou `QTableView`.

Voici un exemple qui utilise `QSqlTableModel` pour réaliser un `SELECT` :

```
QSqlTableModel model;
model.setTable("cd");
model.setFilter("year >= 1998");
model.select();
```

Ce qui est équivalent à la requête

```
SELECT * FROM cd WHERE year >= 1998
```

Pour parcourir un ensemble de résultats, nous récupérons un enregistrement donné au moyen de `QSqlTableModel::record()` et nous accédons aux champs individuels à l'aide de `value()` :

```
for (int i = 0; i < model.rowCount(); ++i) {
```

```
    QSqlRecord record = model.record(i);
    QString title = record.value("title").toString();
    int year = record.value("year").toInt();
    cerr << qPrintable(title) << ": " << year << endl;
}
```

La fonction `QSqlRecord::value()` reçoit soit un nom, soit un index de champ. En travaillant sur des jeux de données de taille importante, il est préférable de désigner les champs par leurs index. Par exemple :

```
int titleIndex = model.record().indexOf("title");
int yearIndex = model.record().indexOf("year");
for (int i = 0; i < model.rowCount(); ++i) {
    QSqlRecord record = model.record(i);
    QString title = record.value(titleIndex).toString();
    int year = record.value(yearIndex).toInt();
    cerr << qPrintable(title) << ": " << year << endl;
}
```

Pour insérer un enregistrement dans une table de base de données, nous utilisons la même approche que pour une insertion dans tout modèle bidimensionnel : en premier lieu, nous appelons `insertRow()` pour créer une nouvelle ligne (enregistrement) vide, puis nous faisons appel à `setData()` pour définir les valeurs de chaque colonne (champ).

```
QSqlTableModel model;
model.setTable("cd");
int row = 0;
model.insertRows(row, 1);
model.setData(model.index(row, 0), 113);
model.setData(model.index(row, 1), "Shanghai My Heart");
model.setData(model.index(row, 2), 224);
model.setData(model.index(row, 3), 2003);
model.submitAll();
```

Après l'appel à `submitAll()`, l'enregistrement peut être déplacé vers un emplacement différent dans la ligne, selon l'organisation de la table. L'appel de cette fonction retournera `false` si l'insertion a échoué.

Une différence importante entre un modèle SQL et un modèle standard est que dans le premier cas, nous devons appeler `submitAll()` pour valider une modification dans la base de données.

Pour mettre à jour un enregistrement, nous devons tout d'abord placer le `QSqlTableModel` sur l'enregistrement à modifier (en exécutant `select()`), par exemple. Nous extrayons ensuite l'enregistrement, mettons à jour les champs voulus, puis réécrivons nos modifications dans la base de données :

```
QSqlTableModel model;
model.setTable("cd");
model.setFilter("id = 125");
model.select();
if (model.rowCount() == 1) {
```

```
    QSqlRecord record = model.record(0);
    record.setValue("title", "Melody A.M.");
    record.setValue("year", record.value("year").toInt() + 1);
    model.setRecord(0, record);
    model.submitAll();
}
```

Si un enregistrement correspond au filtre spécifié, nous le récupérons au moyen de `QSqlTableModel::record()`. Nous appliquons nos modifications et remplaçons l'enregistrement initial par ce dernier.

Comme dans le cas d'un modèle non SQL, il est également possible de réaliser une mise à jour au moyen de `setData()`. Les index de modèle que nous récupérons correspondent à une ligne ou à une colonne donnée :

```
model.select();
if (model.rowCount() == 1) {
    model.setData(model.index(0, 1), "Melody A.M.");
    model.setData(model.index(0, 3),
    model.data(model.index(0, 3)).toInt() + 1);
    model.submitAll();
}
```

La suppression d'un enregistrement est similaire à sa mise à jour :

```
model.setTable("cd");
model.setFilter("id = 125");
model.select();
if (model.rowCount() == 1) {
    model.removeRows(0, 1);
    model.submitAll();
}
```

L'appel de `removeRows()` reçoit le numéro de ligne du premier enregistrement à supprimer ainsi que le nombre d'enregistrements à éliminer. L'exemple suivant supprime tous les enregistrements correspondant au filtre :

```
model.setTable("cd");
model.setFilter("year < 1990");
model.select();
if (model.rowCount() > 0) {
    model.removeRows(0, model.rowCount());
    model.submitAll();
}
```

Les classes `QSqlQuery` et `QSqlTableModel` fournissent une interface entre Qt et une base de données SQL. En utilisant ces classes, nous pouvons créer des formulaires qui présentent les données aux utilisateurs et leur permettent d'insérer, de mettre à jour et de supprimer des enregistrements.

Présenter les données sous une forme tabulaire

Dans de nombreuses situations, il est plus simple de proposer aux utilisateurs une vue tabulaire d'un jeu de données. Dans cette section ainsi que dans la suivante, nous présentons une application CD Collection simple qui fait appel à `QSqlTableModel` et à sa sous-classe `QSqlRelationalTableModel` pour permettre aux utilisateurs d'afficher et d'interagir avec les données stockées dans une base de données.

Le formulaire principal présente une vue maître/détail d'un CD et les pistes du CD en cours de sélection, comme illustré en Figure 13.1.

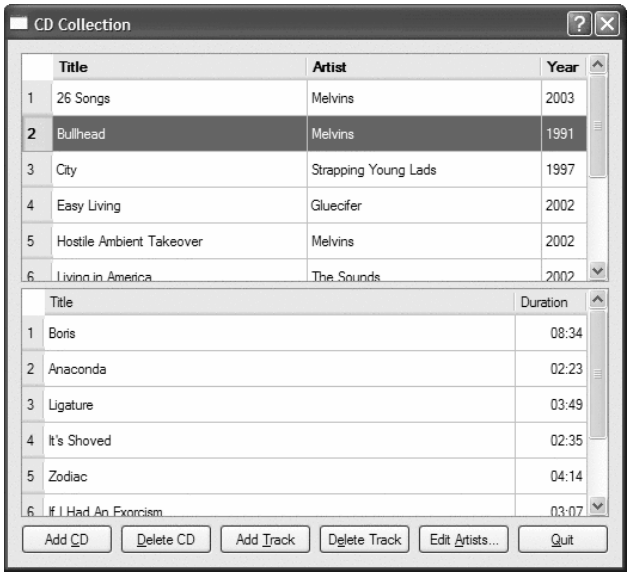


Figure 13.1
L'application CD Collection

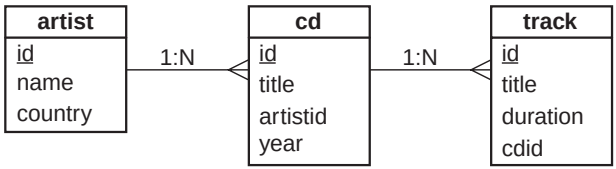
L'application utilise trois tables, définies comme suit :

```
CREATE TABLE artist (
   id INTEGER PRIMARY KEY,
   name VARCHAR(40) NOT NULL,
   country VARCHAR(40));

CREATE TABLE cd (
   id INTEGER PRIMARY KEY,
   title VARCHAR(40) NOT NULL,
   artistid INTEGER NOT NULL,
   year INTEGER NOT NULL,
   FOREIGN KEY (artistid) REFERENCES artist);

CREATE TABLE track (
   id INTEGER PRIMARY KEY,
```

```
    title VARCHAR(40) NOT NULL,
    duration INTEGER NOT NULL,
    cdid INTEGER NOT NULL,
    FOREIGN KEY (cdid) REFERENCES cd);
```

Certaines bases de données ne supportent pas les clés étrangères. Dans ce cas, nous devons supprimer les clauses `FOREIGN KEY`. L'exemple fonctionnera toujours, mais la base de données n'appliquera pas l'intégrité référentielle. (Voir Figure 13.2)

Figure 13.2
Les tables de l'application CD Collection



Dans cette section, nous allons écrire une boîte de dialogue dans laquelle les utilisateurs pourront modifier une liste d'artistes en utilisant une forme tabulaire simple. L'utilisateur peut insérer ou supprimer des artistes au moyen des boutons du formulaire. Les mises à jour peuvent être appliquées directement, simplement en modifiant le texte des cellules. Les changements sont appliqués à la base de données lorsque l'utilisateur appuie sur Entrée ou passe à un autre enregistrement. (Voir Figure 13.3)

Figure 13.3
La boîte de dialogue ArtistForm

Voici la définition de classe pour cette boîte de dialogue :

```
class ArtistForm : public QDialog
{
   Q_OBJECT

public:
   ArtistForm(const QString &name, QWidget *parent = 0);

private slots:
   void addArtist();
   void deleteArtist();
```

```
    void beforeInsertArtist(QSqlRecord &record);

private:
    enum {
        Artist_Id = 0,
        Artist_Name = 1,
        Artist_Country = 2
    };

    QSqlTableModel *model;
    QTableView *tableView;
    QPushButton *addButton;
    QPushButton *deleteButton;
    QPushButton *closeButton;
};
```

Le constructeur est très similaire à celui qui serait utilisé pour créer un formulaire basé sur un modèle non SQL :

```
ArtistForm::ArtistForm(const QString &name, QWidget *parent)
    : QDialog(parent)
{
    model = new QSqlTableModel(this);
    model->setTable("artist");
    model->setSort(Artist_Name, Qt::AscendingOrder);
    model->setHeaderData(Artist_Name, Qt::Horizontal, tr("Name"));
    model->setHeaderData(Artist_Country, Qt::Horizontal, tr("Country"));
    model->select();
    connect(model, SIGNAL(beforeInsert(QSqlRecord &)),
        this, SLOT(beforeInsertArtist(QSqlRecord &)));

    tableView = new QTableView;
    tableView->setModel(model);
    tableView->setColumnHidden(Artist_Id, true);
    tableView->setSelectionBehavior(QAbstractItemView::SelectRows);
    tableView->resizeColumnsToContents();

    for (int row = 0; row < model->rowCount(); ++row) {
        QSqlRecord record = model->record(row);
        if (record.value(Artist_Name).toString() == name) {
            tableView->selectRow(row);
            break;
        }
    }
    ...

}
```

Nous commençons le constructeur en créant un `QSqlTableModel`. Nous lui transmettons `this` comme parent pour octroyer la propriété au formulaire. Nous avons choisi de baser le tri sur la colonne 1 (spécifiée par la constante `Artist_Name`), ce qui correspond au champ `name`. Si nous ne spécifions pas d'en-tête de colonnes, ce sont les noms des champs qui sont utilisés.

Nous préférons les nommer nous-mêmes pour garantir une casse et une internationalisation correctes.

Nous créons ensuite un `QTableView` pour visualiser le modèle. Nous masquons le champ `id` et définissons les largeurs des colonnes en fonction de leur texte afin de ne pas avoir à afficher de points de suspension.

Le constructeur de `ArtistForm` reçoit le nom de l'artiste qui doit être sélectionné à l'ouverture de la boîte de dialogue. Nous parcourons les enregistrements de la table `artist` et sélectionnons l'artiste voulu. Le reste du code du constructeur permet de créer et de connecter les boutons ainsi que de positionner les widgets enfants.

```
void ArtistForm::addArtist()
{
   int row = model->rowCount();
   model->insertRow(row);
   QModelIndex index = model->index(row, Artist_Name);
   tableView->setCurrentIndex(index);
   tableView->edit(index);
}
```

Pour ajouter un nouvel artiste, nous insérons une ligne vierge dans le bas de `QTableView`. L'utilisateur peut maintenant entrer le nom et le pays du nouvel artiste. S'il confirme l'insertion en appuyant sur Entrée, le signal `beforeInsert()` est émis puis le nouvel enregistrement est inséré dans la base de données.

```
void ArtistForm::beforeInsertArtist(QSqlRecord &record)
{
   record.setValue("id", generateId("artist"));
}
```

Dans le constructeur, nous avons connecté le signal `beforeInsert()` du modèle à ce slot. Une référence non-const à l'enregistrement nous est transmise juste avant son insertion dans la base de données. A ce stade, nous remplissons son champ `id`.

Comme nous aurons besoin de `generateId()` à plusieurs reprises, nous la définissons "en ligne" dans un fichier d'en-tête et l'incluons à chaque fois que nécessaire. Voici un moyen rapide (et inefficace) de l'implémenter :

```
inline int generateId(const QString &table)
{
   QSqlQuery query;
   query.exec("SELECT MAX(id) FROM " + table);
   int id = 0;
   if (query.next())
      id = query.value(0).toInt() + 1;
   return id;
}
```

La fonction `generateId()` n'est assurée de fonctionner correctement que si elle est exécutée dans le contexte de la même transaction que l'instruction `INSERT` correspondante.

Certaines bases de données supportent les champs générés automatiquement, et il est généralement nettement préférable d'utiliser la prise en charge spécifique à chaque base de données pour cette opération.

La dernière possibilité offerte par la boîte de dialogue `ArtistForm` est la suppression. Au lieu d'effectuer des suppressions en cascade (que nous avons abordées brièvement), nous avons choisi de n'autoriser que les suppressions d'artistes ne possédant pas de CD dans la collection.

```
void ArtistForm::deleteArtist()
{
   tableView->setFocus();
   QModelIndex index = tableView->currentIndex();
   if (!index.isValid())
       return;
   QSqlRecord record = model->record(index.row());

   QSqlTableModel cdModel;
   cdModel.setTable("cd");
   cdModel.setFilter("artistid = " + record.value("id").toString());
   cdModel.select();
   if (cdModel.rowCount() == 0) {
       model->removeRow(tableView->currentIndex().row());
   } else {
      QMessageBox::information(this,
         tr("Delete Artist"),
         tr("Cannot delete %1 because there are CDs associated "
            "with this artist in the collection.")
         .arg(record.value("name").toString()));
   }
}
```

Si un enregistrement est sélectionné, nous déterminons si l'artiste possède un CD. Si tel n'est pas le cas, nous le supprimons immédiatement. Sinon, nous affichons une boîte de message expliquant pourquoi la suppression n'a pas eu lieu. Strictement parlant, nous aurions dû utiliser une transaction, car tel que le code se présente, il est possible que l'artiste que nous supprimons soit associé à un CD entre les appels de `cdModel.select()` et `model->removeRow()`. Nous présenterons une transaction dans la prochaine section.

Implémenter des formulaires maître/détail

A présent, nous allons réviser le formulaire principal avec une approche maître/détail. La vue maître est une liste de CD. La vue détail est une liste de pistes pour le CD en cours. Ce formulaire représente la fenêtre principale de l'application CD Collection comme illustré en Figure 13.1.

```
class MainForm : public QWidget
{
   Q_OBJECT
```

```
public:
   MainForm();

private slots:
   void addCd();
   void deleteCd();
   void addTrack();
   void deleteTrack();
   void editArtists();
   void currentCdChanged(const QModelIndex &index);
   void beforeInsertCd(QSqlRecord &record);
   void beforeInsertTrack(QSqlRecord &record);
   void refreshTrackViewHeader();

private:
   enum {
      Cd_Id = 0,
      Cd_Title = 1,
      Cd_ArtistId = 2,
      Cd_Year = 3
   };

   enum {
      Track_Id = 0,
      Track_Title = 1,
      Track_Duration = 2,
      Track_CdId = 3
   };

   QSqlRelationalTableModel *cdModel;
   QSqlTableModel *trackModel;
   QTableView *cdTableView;
   QTableView *trackTableView;
   QPushButton *addCdButton;
   QPushButton *deleteCdButton;
   QPushButton *addTrackButton;
   QPushButton *deleteTrackButton;
   QPushButton *editArtistsButton;
   QPushButton *quitButton;
};
```

Au lieu d'un `QSqlTableModel`, nous utilisons un `QSqlRelationalTableModel` pour la table `cd`, car nous devons gérer les clés étrangères. Nous allons maintenant revoir chaque fonction tour à tour, en commençant par le constructeur que nous étudierons par segments car il est assez long.

```
MainForm::MainForm()
{
   cdModel = new QSqlRelationalTableModel(this);
   cdModel->setTable("cd");
   cdModel->setRelation(Cd_ArtistId,
                        QSqlRelation("artist", "id", "name"));
   cdModel->setSort(Cd_Title, Qt::AscendingOrder);
```

```
    cdModel->setHeaderData(Cd_Title, Qt::Horizontal, tr("Title"));
    cdModel->setHeaderData(Cd_ArtistId, Qt::Horizontal, tr("Artist"));
    cdModel->setHeaderData(Cd_Year, Qt::Horizontal, tr("Year"));
    cdModel->select();
```

Le constructeur définit tout d'abord le `QSqlRelationalTableModel` qui gère la table `cd`. L'appel de `setRelation()` indique au modèle que son champ `artistid` (dont l'index est inclus dans `Cd_ArtistId`) possède la clé étrangère `id` de la table `artist`, et que le contenu du champ `name` correspondant doit être affiché à la place des ID. Si l'utilisateur choisit d'éditer ce champ (par exemple en appuyant sur F2), le modèle présentera automatiquement une zone de liste déroulante avec les noms de tous les artistes, et si l'utilisateur choisit un artiste différent, il mettra la table `cd` à jour.

```
cdTableView = new QTableView;
cdTableView->setModel(cdModel);
cdTableView->setItemDelegate(new QSqlRelationalDelegate(this));
cdTableView->setSelectionMode(QAbstractItemView::SingleSelection);
cdTableView->setSelectionBehavior(QAbstractItemView::SelectRows);
cdTableView->setColumnHidden(Cd_Id, true);
cdTableView->resizeColumnsToContents();
```

La définition de la vue pour la table `cd` est similaire à ce que nous avons déjà vu. La seule différence significative est la suivante : au lieu d'utiliser le délégué par défaut de la vue, nous utilisons `QSqlRelationalDelegate`. C'est ce délégué qui gère les clés étrangères.

```
trackModel = new QSqlTableModel(this);
trackModel->setTable("track");
trackModel->setHeaderData(Track_Title, Qt::Horizontal, tr("Title"));
trackModel->setHeaderData(Track_Duration, Qt::Horizontal,
                          tr("Duration"));

trackTableView = new QTableView;
trackTableView->setModel(trackModel);
trackTableView->setItemDelegate(
             new TrackDelegate(Track_Duration, this));
trackTableView->setSelectionMode(
            QAbstractItemView::SingleSelection);
trackTableView->setSelectionBehavior(QAbstractItemView::SelectRows);
```

Pour ce qui est des pistes, nous n'allons montrer que leurs noms et leurs durées. C'est pourquoi `QSqlTableModel` est suffisant. Le seul aspect remarquable de cette partie du code est que nous utilisons le `TrackDelegate` développé dans le Chapitre 10 pour afficher les durées des pistes sous la forme "*minutes:secondes*" et permettre leur édition en utilisant un `QTimeEdit` adapté.

La création, la connexion et la disposition des vues ainsi que des boutons ne présente pas de surprise. C'est pourquoi la seule autre partie du constructeur que nous allons présenter contient quelques connexions non évidentes.

```
...
connect(cdTableView->selectionModel(),
```

```
        SIGNAL(currentRowChanged(const QModelIndex &,
                                 const QModelIndex &)),
        this, SLOT(currentCdChanged(const QModelIndex &)));
connect(cdModel, SIGNAL(beforeInsert(QSqlRecord &)),
        this, SLOT(beforeInsertCd(QSqlRecord &)));
connect(trackModel, SIGNAL(beforeInsert(QSqlRecord &)),
        this, SLOT(beforeInsertTrack(QSqlRecord &)));
connect(trackModel, SIGNAL(rowsInserted(const QModelIndex &, int,
                                        int)),
        this, SLOT(refreshTrackViewHeader()));
...
}
```

La première connexion est inhabituelle, car au lieu de connecter un widget, nous établissons une connexion avec un modèle de sélection. La classe `QItemSelectionModel` est utilisée pour assurer le suivi des sélections dans les vues. En étant connecté au modèle de sélection de la vue table, notre slot `currentCdChanged()` sera appelé dès que l'utilisateur passe d'un enregistrement à l'autre.

```
void MainForm::currentCdChanged(const QModelIndex &index)
{
   if (index.isValid()) {
       QSqlRecord record = cdModel->record(index.row());
       int id = record.value("id").toInt();
       trackModel->setFilter(QString("cdid = %1").arg(id));
   } else {
       trackModel->setFilter("cdid = -1");
   }
   trackModel->select();
   refreshTrackViewHeader();
}
```

Ce slot est appelé dès que le CD en cours change, ce qui se produit lorsque l'utilisateur passe à un autre CD (en cliquant ou en utilisant les touches fléchées). Si le CD est invalide (s'il n'existe pas de CD, si un nouveau CD est en cours d'insertion ou encore si celui en cours vient d'être supprimé), nous définissons le `cdid` de la table `track` en `1` (un ID invalide qui ne correspondra à aucun enregistrement).

Puis, en ayant défini le filtre, nous sélectionnons les enregistrements de piste correspondants. La fonction `refreshTrackViewHeader()` sera étudiée dans un moment.

```
void MainForm::addCd()
{
   int row = 0;
   if (cdTableView->currentIndex().isValid())
       row = cdTableView->currentIndex().row();

   cdModel->insertRow(row);
   cdModel->setData(cdModel->index(row, Cd_Year),
                    QDate::currentDate().year());
```

```
    QModelIndex index = cdModel->index(row, Cd_Title);
    cdTableView->setCurrentIndex(index);
    cdTableView->edit(index);
}
```

Lorsque l'utilisateur clique sur le bouton Add CD, une nouvelle ligne vierge est insérée dans le `cdTableView` et nous entrons en mode édition. Nous définissons également une valeur par défaut pour le champ `year`. A ce stade, l'utilisateur peut modifier l'enregistrement en remplissant les champs vierges et en sélectionnant un artiste dans la zone de liste déroulante qui est automatiquement fournie par le `QSqlRelationalTableModel` grâce à l'appel de `setRelation(). Il peut aussi` modifier l'année si celle proposée par défaut s'avère inappropriée. Si l'utilisateur confirme l'insertion en appuyant sur Entrée, l'enregistrement est inséré. L'utilisateur peut annuler en appuyant sur Echap.

```
void MainForm::beforeInsertCd(QSqlRecord &record)
{
    record.setValue("id", generateId("cd"));
}
```

Ce slot est appelé lorsque le `cdModel` émet son signal `beforeInsert()`. Nous l'utilisons pour remplir le champ `id` de la même façon que nous l'avons fait pour insérer de nouveaux artistes. Les mêmes règles s'appliquent : cette opération doit s'effectuer dans la portée d'une transaction et avec les méthodes de création d'ID spécifiques à la base de données (par exemple, les ID générés automatiquement).

```
void MainForm::deleteCd()
{
    QModelIndex index = cdTableView->currentIndex();
    if (!index.isValid())
        return;
    QSqlDatabase db = QSqlDatabase::database();
    db.transaction();
    QSqlRecord record = cdModel->record(index.row());
    int id = record.value(Cd_Id).toInt();
    int tracks = 0;
    QSqlQuery query;
    query.exec(QString("SELECT COUNT(*) FROM track WHERE cdid = %1")
               .arg(id));
    if (query.next())
        tracks = query.value(0).toInt();
    if (tracks > 0) {
        int r = QMessageBox::question(this, tr("Delete CD"),
                        tr("Delete \"%1\" and all its tracks?")
                        .arg(record.value(Cd_ArtistId).toString()),
                        QMessageBox::Yes | QMessageBox::Default,
                        QMessageBox::No | QMessageBox::Escape);
       if (r == QMessageBox::No) {
           db.rollback();
           return;
       }
```

```
        query.exec(QString("DELETE FROM track WHERE cdid = %1")
                   .arg(id));
    }
    cdModel->removeRow(index.row());
    cdModel->submitAll();
    db.commit();

    currentCdChanged(QModelIndex());
}
```

Lorsque l'utilisateur clique sur le bouton Delete CD, ce slot est appelé. Quand un CD est en cours, nous déterminons son nombre de pistes. Si nous ne trouvons pas de piste, nous supprimons directement l'enregistrement du CD. S'il existe au moins une piste, nous demandons à l'utilisateur de confirmer la suppression. S'il clique sur Yes, nous supprimons tous les enregistrements de piste, puis l'enregistrement du CD. Toutes ces opérations sont effectuées dans la portée d'une transaction. Ainsi, soit la suppression en cascade échoue en bloc, soit elle réussit dans son ensemble – en supposant que la base de données en question supporte les transactions.

La gestion des données de piste est très similaire à celle des données de CD. Les mises à jour peuvent être effectuées simplement *via* les cellules d'édition fournies à l'utilisateur. Dans le cas des durées de piste, notre `TrackDelegate` s'assure qu'elles sont présentées dans le bon format et qu'elles sont facilement modifiables au moyen de `QTimeEdit`.

```
void MainForm::addTrack()
{
    if (!cdTableView->currentIndex().isValid())
        return;

    int row = 0;
    if (trackTableView->currentIndex().isValid())
        row = trackTableView->currentIndex().row();

    trackModel->insertRow(row);
    QModelIndex index = trackModel->index(row, Track_Title);
    trackTableView->setCurrentIndex(index);
    trackTableView->edit(index);
}
```

Le fonctionnement ici est le même que celui de `addCd()`, avec une nouvelle ligne vierge insérée dans la vue.

```
void MainForm::beforeInsertTrack(QSqlRecord &record)
{
    QSqlRecord cdRecord = cdModel->record(cdTableView->currentIndex()
                                                      .row());
    record.setValue("id", generateId("track"));
    record.setValue("cdid", cdRecord.value(Cd_Id).toInt());
}
```

Si l'utilisateur confirme l'insertion initiée par `addTrack()`, cette fonction est appelée pour remplir les champs `id` et `cdid`. Les règles mentionnées précédemment s'appliquent bien sûr toujours ici.

```
void MainForm::deleteTrack()
{
   trackModel->removeRow(trackTableView->currentIndex().row());
   if (trackModel->rowCount() == 0)
       trackTableView->horizontalHeader()->setVisible(false);
}
```

Si l'utilisateur clique sur le bouton `Delete Track`, nous supprimons la piste sans formalité. Il serait facile d'afficher une boîte de message de type Yes/No si nous envisagions de faire confirmer les suppressions.

```
void MainForm::refreshTrackViewHeader()
{
   trackTableView->horizontalHeader()->setVisible(
           trackModel->rowCount() > 0);
   trackTableView->setColumnHidden(Track_Id, true);
   trackTableView->setColumnHidden(Track_CdId, true);
   trackTableView->resizeColumnsToContents();
}
```

Le slot `refreshTrackViewHeader()` est invoqué depuis plusieurs emplacements pour s'assurer que l'en-tête horizontal de la vue de piste n'est présenté que s'il existe des pistes à afficher. Il masque aussi les champs `id` et `cdid` et redimensionne les colonnes de table visibles en fonction du contenu courant de la table.

```
void MainForm::editArtists()
{
   QSqlRecord record = cdModel->record(cdTableView->currentIndex()
                                       .row());
   ArtistForm artistForm(record.value(Cd_ArtistId).toString(), this);
   artistForm.exec();
   cdModel->select();
}
```

Ce slot est appelé si l'utilisateur clique sur le bouton `Edit Artists`. Il affiche les données concernant l'artiste du CD en cours, invoquant le `ArtistForm` traité dans la section précédente et sélectionnant l'artiste approprié. S'il n'existe pas d'enregistrement en cours, un enregistrement vide est retourné par `record()`. Il ne correspond naturellement à aucun artiste dans le formulaire. Voici ce qui se produit véritablement : comme nous utilisons un `QSqlRelationalTableModel` qui établit une correspondance entre les ID des artistes et leurs noms, la valeur qui est retournée lorsque nous appelons `record.value(Cd_ArtistId)` est le nom de l'artiste (qui sera une chaîne vide si l'enregistrement est vide). Nous forçons enfin le `cdModel` à sélectionner de nouveau ses données, ce qui conduit le `cdTableView` à rafraîchir ses cellules visibles.

Cette opération permet de s'assurer que les noms des artistes sont affichés correctement, certains d'entre eux ayant pu être modifiés par l'utilisateur dans la boîte de dialogue `ArtistForm`.

Pour les projets qui utilisent les classes SQL, nous devons ajouter la ligne

```
QT              += sql
```

aux fichiers `.pro`, ce qui garantit la liaison de l'application à la bibliothèque *QtSql*.

Ce chapitre vous a démontré que les classes vue/modèle de Qt facilitent autant que possible l'affichage et la modification de données dans les bases SQL. Dans les cas où les clés étrangères se réfèrent à des tables comportant de nombreux enregistrements (des milliers, voir plus), il est probablement préférable de créer votre propre délégué et de l'utiliser pour présenter un formulaire de "listes de valeurs" offrant des possibilités de recherche au lieu de vous reposer sur les zones de liste déroulante par défaut de `QSqlRelationalTableModel`. Et si nous souhaitons présenter des enregistrements en utilisant un mode formulaire, nous devons le gérer par nous-mêmes : en faisant appel à un `QSqlQuery` ou à un `QSqlTableModel` pour gérer l'interaction avec la base de données, et en établissant une correspondance entre le contenu des widgets de l'interface utilisateur que nous souhaitons utiliser pour présenter et modifier les données et la base de données concernée dans notre propre code.

14

Gestion de réseau

Au sommaire de ce chapitre

- ✔ Programmer les clients FTP
- ✔ Programmer les clients HTTP
- ✔ Programmer les applications client/serveur TCP
- ✔ Envoyer et recevoir des datagrammes UDP

Qt fournit les classes `QFtp` et `QHttp` pour la programmation de FTP et HTTP. Ces protocoles sont faciles à utiliser pour télécharger des fichiers et, dans le cas de HTTP, pour envoyer des requêtes aux serveurs Web et récupérer les résultats.

Qt fournit également les classes de bas niveau `QTcpSocket` et `QUdpSocket`, qui implémentent les protocoles de transport TCP et UDP. TCP est un protocole orienté connexion fiable qui agit en termes de flux de données transmis entre les nœuds réseau, alors que UDP est un protocole fonctionnant en mode non connecté non fiable qui permet d'envoyer des paquets discrets entre des nœuds réseau. Tous deux peuvent être utilisés pour créer des applications réseau clientes et serveur. En ce qui concerne les serveurs, nous avons aussi besoin de la classe `QTcpServer` pour gérer les connexions TCP entrantes.

Programmer les clients FTP

La classe `QFtp` implémente le côté client du protocole FTP dans Qt. Elle offre diverses fonctions destinées à réaliser les opérations FTP les plus courantes et nous permet d'exécuter des commandes FTP arbitraires.

La classe `QFtp` fonctionne de façon asynchrone. Lorsque nous appelons une fonction telle que `get()` ou `put()`, elle se termine immédiatement et le transfert de données se produit quand le contrôle revient à la boucle d'événement de Qt. Ainsi, l'interface utilisateur reste réactive pendant l'exécution des commandes FTP.

Nous allons commencer par un exemple qui illustre comment récupérer un fichier unique au moyen de `get()`. L'exemple est une application de console nommée `ftpget` qui télécharge le fichier distant spécifié sur la ligne de commande. Commençons par la fonction `main()` :

```
int main(int argc, char *argv[])
{
   QCoreApplication app(argc, argv);
   QStringList args = app.arguments();

   if (args.count() != 2) {
       cerr << "Usage: ftpget url" << endl
         << "Example:" << endl
         << " ftpget ftp://ftp.trolltech.com/mirrors" << endl;
      return 1;
   }

   FtpGet getter;
   if (!getter.getFile(QUrl(args[1])))
       return 1;

   QObject::connect(&getter, SIGNAL(done()), &app, SLOT(quit()));

   return app.exec();
}
```

Nous créons un `QCoreApplication` plutôt que sa sous-classe `QApplication` pour éviter une liaison dans la bibliothèque *QtGui*. La fonction `QCoreApplication::arguments()` retourne les arguments de ligne de commande sous forme de `QStringList`, le premier élément étant le nom sous lequel le programme a été invoqué, et tous les arguments propres à Qt (tels que `-style`) étant supprimés. Le cœur de la fonction `main()` est la construction de l'objet `FtpGet` et l'appel de `getFile()`. Si l'appel réussit, nous laissons la boucle d'événement s'exécuter jusqu'à la fin du téléchargement.

Tout le travail est effectué par la sous-classe `FtpGet`, qui est définie comme suit :

```
class FtpGet : public QObject
{
   Q_OBJECT
```

```
public:
   FtpGet(QObject *parent = 0);

   bool getFile(const QUrl &url);

signals:
   void done();

private slots:
   void ftpDone(bool error);

private:
   QFtp ftp;
   QFile file;
};
```

La classe possède une fonction publique, `getFile()`, qui récupère le fichier spécifié par une URL. La classe `QUrl` fournit une interface de haut niveau destinée à extraire les différents segments d'une URL, tels que le nom du fichier, le chemin d'accès, le protocole et le port.

`FtpGet` possède un slot privé, `ftpDone()`, qui est appelé lorsque le transfert de fichier est terminé, et un signal `done()` qui est émis une fois le fichier téléchargé. La classe contient également deux variables privées : la variable `ftp`, de type `QFtp`, qui encapsule la connexion avec un serveur FTP et la variable `file` qui est utilisée pour écrire le fichier téléchargé sur le disque.

```
FtpGet::FtpGet(QObject *parent)
   : QObject(parent)
{
   connect(&ftp, SIGNAL(done(bool)), this, SLOT(ftpDone(bool)));
}
```

Dans le constructeur, nous connectons le signal `QFtp::done(bool)` à notre slot privé `ftpDone(bool)`. `QFtp` émet `done(bool)` une fois le traitement de toutes les requêtes terminé. Le paramètre `bool` indique si une erreur s'est produite ou non.

```
bool FtpGet::getFile(const QUrl &url)
{
   if (!url.isValid()) {
       cerr << "Error: Invalid URL" << endl;
       return false;
   }

   if (url.scheme() != "ftp") {
       cerr << "Error: URL must start with 'ftp:'" << endl;
       return false;
   }

   if (url.path().isEmpty()) {
       cerr << "Error: URL has no path" << endl;
       return false;
   }
```

```
    QString localFileName = QFileInfo(url.path()).fileName();
    if (localFileName.isEmpty())
        localFileName = "ftpget.out";

    file.setFileName(localFileName);
    if (!file.open(QIODevice::WriteOnly)) {
        cerr << "Error: Cannot open " << qPrintable(file.fileName())
             << " for writing: " << qPrintable(file.errorString())
             << endl;
        return false;
    }

    ftp.connectToHost(url.host(), url.port(21));
    ftp.login();
    ftp.get(url.path(), &file);
    ftp.close();
    return true;
}
```

La fonction `getFile()` commence en vérifiant l'URL transmise. Si un problème est rencontré, elle émet un message d'erreur vers `cerr` et retourne `false` pour indiquer que le téléchargement a échoué.

Au lieu d'obliger l'utilisateur à créer un nom de fichier local, nous essayons de générer un nom judicieux constitué de l'URL elle-même, avec `ftpget.out` comme solution de secours. Si nous ne parvenons pas à ouvrir le fichier, nous affichons un message d'erreur et retournons `false`.

Nous exécutons ensuite une séquence de quatre commandes FTP en utilisant notre objet `QFtp`. L'appel de `url.port(21)` retourne le numéro de port mentionné dans l'URL, ou le port 21 si l'URL n'en spécifie aucun. Aucun nom d'utilisateur ou mot de passe n'étant transmis à la fonction `login()`, on tente une ouverture de session anonyme. Le second argument de `get()` spécifie le périphérique de sortie.

Les commandes FTP sont placées en file d'attente et exécutées dans la boucle d'événement de Qt. L'achèvement de toutes les commandes est indiqué par le signal `done(bool)` de `QFtp`, que nous avons connecté à `ftpDone(bool)` dans le constructeur.

```
void FtpGet::ftpDone(bool error)
{
    if (error) {
        cerr << "Error: " << qPrintable(ftp.errorString()) << endl;
    } else {
        cerr << "File downloaded as " << qPrintable(file.fileName())
             << endl;
    }
    file.close();
    emit done();
}
```

Les commandes FTP ayant toutes été exécutées, nous fermons le fichier et émettons notre propre signal `done()`. Il peut sembler étrange de fermer le fichier ici, et non après l'appel de `ftp.close()` à la fin de la fonction `getFile()`, mais souvenez-vous que les commandes FTP sont exécutées de façon asynchrone et peuvent très bien être en cours à la fin de l'exécution de `getFile()`. Seule l'émission du signal `done()` de l'objet `QFtp` nous permet de savoir que le téléchargement est terminé et que le fichier peut être fermé en toute sécurité.

`QFtp` fournit plusieurs commandes FTP, dont `connectToHost()`, `login()`, `close()`, `list()`, `cd()`, `get()`, `put()`, `remove()`, `mkdir()`, `rmdir()` et `rename()`. Toutes ces fonctions programment une commande FTP et retournent un numéro d'ID qui identifie cette commande. Il est également possible de contrôler le mode et le type de transfert (l'option par défaut est un mode passif et un type binaire).

Les commandes FTP arbitraires peuvent être exécutées au moyen de la commande `rawCommand()`. Voici, par exemple, comment exécuter une commande `SITE CHMOD` :

```
ftp.rawCommand("SITE CHMOD 755 fortune");
```

`QFtp` émet le signal `commandStarted(int)` quand il commence à exécuter une commande et le signal `commandFinished(int, bool)` une fois la commande terminée. Le paramètre `int` est le numéro d'ID qui identifie la commande. Si nous nous intéressons au sort des commandes individuelles, nous pouvons stocker les numéros d'ID lors de la programmation des commandes. Le fait de suivre ces numéros nous permet de fournir un rapport détaillé à l'utilisateur. Par exemple :

```
bool FtpGet::getFile(const QUrl &url)
{

   ...
   connectId = ftp.connectToHost(url.host(), url.port(21));
   loginId = ftp.login();
   getId = ftp.get(url.path(), &file);
   closeId = ftp.close();
   return true;
}

void FtpGet::ftpCommandStarted(int id)
{
   if (id == connectId) {
       cerr << "Connecting..." << endl;
   } else if (id == loginId) {
       cerr << "Logging in..." << endl;
   ...
}
```

Un autre moyen de fournir un rapport consiste à établir une connexion au signal `stateChanged()` de `QFtp`, qui est émis lorsque la connexion entre dans un nouvel état (`QFtp::Connecting`, `QFtp::Connected`, `QFtp::LoggedIn`, etc.)

Dans la plupart des applications, nous nous intéressons plus au sort de la séquence de commandes dans son ensemble qu'aux commandes particulières. Dans ce cas, nous pouvons simplement nous connecter au signal `done(bool)`, qui est émis dès que la file d'attente de commandes est vide.

Quand une erreur se produit, `QFtp` vide automatiquement la file d'attente de commandes. Ainsi, si la connexion ou l'ouverture de session échoue, les commandes qui suivent dans la file d'attente ne sont jamais exécutées. Si nous programmons de nouvelles commandes au moyen de l'objet `QFtp` après que l'erreur se soit produite, elles sont placées en file d'attente et exécutées.

Dans le fichier `.pro` de l'application, la ligne suivante est nécessaire pour établir une liaison avec la bibliothèque *QtNetwork* :

```
QT            += network
```

Nous allons maintenant étudier un exemple plus sophistiqué. Le programme de ligne de commande `spider` télécharge tous les fichiers situés dans un répertoire FTP. La logique réseau est gérée dans la classe `Spider` :

```
class Spider : public QObject
{
   Q_OBJECT

public:
   Spider(QObject *parent = 0);

   bool getDirectory(const QUrl &url);

signals:
   void done();

private slots:
   void ftpDone(bool error);
   void ftpListInfo(const QUrlInfo &urlInfo);

private:
   void processNextDirectory();
   QFtp ftp;
   QList<QFile *> openedFiles;
   QString currentDir;
   QString currentLocalDir;
   QStringList pendingDirs;
};
```

Le répertoire de départ est spécifié en tant que `QUrl` et est défini au moyen de la fonction `getDirectory()`.

```
Spider::Spider(QObject *parent)
   : QObject(parent)
{
```

```
    connect(&ftp, SIGNAL(done(bool)), this, SLOT(ftpDone(bool)));
    connect(&ftp, SIGNAL(listInfo(const QUrlInfo &)),
            this, SLOT(ftpListInfo(const QUrlInfo &)));
}
```

Dans le constructeur, nous établissons deux connexions signal/slot. Le signal `listInfo (const QUrlInfo &)` est émis par `QFtp` lorsque nous demandons un listing de répertoires (dans `getDirectory()`) pour chaque fichier récupéré. Ce signal est connecté à un slot nommé `ftpListInfo()`, qui télécharge le fichier associé à l'URL qui lui est fournie.

```
bool Spider::getDirectory(const QUrl &url)
{
    if (!url.isValid()) {
        cerr << "Error: Invalid URL" << endl;
        return false;
    }

    if (url.scheme() != "ftp") {
        cerr << "Error: URL must start with 'ftp:'" << endl;
        return false;
    }

    ftp.connectToHost(url.host(), url.port(21));
    ftp.login();

    QString path = url.path();
    if (path.isEmpty())
        path = "/";

    pendingDirs.append(path);
    processNextDirectory();

    return true;
}
```

Lorsque la fonction `getDirectory()` est appelée, elle commence par effectuer quelques vérifications de base, et, si tout va bien, tente d'établir une connexion FTP. Elle appelle `processNextDirectory()` pour lancer le téléchargement du répertoire racine.

```
void Spider::processNextDirectory()
{
    if (!pendingDirs.isEmpty()) {
        currentDir = pendingDirs.takeFirst();
        currentLocalDir = "downloads/" + currentDir;
        QDir(".").mkpath(currentLocalDir);

        ftp.cd(currentDir);
        ftp.list();
    } else {
        emit done();
    }
}
```

La fonction `processNextDirectory()` reçoit le premier répertoire distant provenant de la liste `pendingDirs` et crée un répertoire correspondant dans le système de fichiers local. Elle indique ensuite à l'objet `QFtp` de remplacer le répertoire existant par celui reçu et de répertorier ses fichiers. Pour tout fichier traité par `list()`, un signal `listInfo()` provoquant l'appel du slot `ftpListInfo()` est émis.

S'il ne reste plus de répertoire à traiter, la fonction émet le signal `done()` pour indiquer que le téléchargement est achevé.

```
void Spider::ftpListInfo(const QUrlInfo &urlInfo)
{
   if (urlInfo.isFile()) {
      if (urlInfo.isReadable()) {
         QFile *file = new QFile(currentLocalDir + "/"
                                 + urlInfo.name());

         if (!file->open(QIODevice::WriteOnly)) {
             cerr << "Warning: Cannot open file "
                  << qPrintable(
                     QDir::convertSeparators(file->fileName()))
                << endl;
             return;
         }

         ftp.get(urlInfo.name(), file);
         openedFiles.append(file);
      }
   } else if (urlInfo.isDir() && !urlInfo.isSymLink()) {
       pendingDirs.append(currentDir + "/" + urlInfo.name());
   }
}
```

Le paramètre `urlInfo` du slot `ftpListInfo()` fournit des informations détaillées concernant un fichier distant. S'il s'agit d'un fichier normal (et non d'un répertoire) lisible, nous appelons `get()` pour le télécharger. L'objet `QFile` utilisé pour le téléchargement est alloué au moyen de `new` et un pointeur dirigé vers celui-ci est stocké dans la liste `openedFiles`.

Si le `QUrlInfo` contient les détails d'un répertoire distant qui n'est pas un lien symbolique, nous ajoutons ce répertoire à la liste `pendingDirs`. Nous ignorons les liens symboliques car ils peuvent aisément mener à une récurrence infinie.

```
void Spider::ftpDone(bool error)
{
   if (error) {
       cerr << "Error: " << qPrintable(ftp.errorString()) << endl;
   } else {
       cout << "Downloaded " << qPrintable(currentDir) << " to "
            << qPrintable(QDir::convertSeparators(
                             QDir(currentLocalDir).canonicalPath()));
   }
```

```
    qDeleteAll(openedFiles);
    openedFiles.clear();

    processNextDirectory();
}
```

Le slot `ftpDone()` est appelé lorsque toutes les commandes FTP sont terminées ou si une erreur se produit. Nous supprimons les objets `QFile` pour éviter les fuites de mémoire et également pour fermer chaque fichier. Nous appelons enfin `processNextDirectory()`. S'il reste des répertoires, tout le processus recommence pour le répertoire suivant dans la liste. Dans le cas contraire, le téléchargement est interrompu une fois `done()` émis.

Si aucune erreur n'intervient, la séquence de signaux et de commandes FTP est la suivante :

```
connectToHost(host, port)
login()

cd(directory_1)
list()
   emit listInfo(file_1_1)
      get(file_1_1)
   emit listInfo(file_1_2)
      get(file_1_2)
   ...
emit done()

...

cd(directory_N)
list()
   emit listInfo(file_N_1)
      get(file_N_1)
   emit listInfo(file_N_2)
      get(file_N_2)
   ...
emit done()
```

Si un fichier est un répertoire, il est ajouté à la liste `pendingDirs`. Une fois le dernier fichier de la commande `list()` téléchargé, une nouvelle commande `cd()` est émise, suivie d'une commande `list()` avec le répertoire suivant en attente. Tout le processus recommence alors avec ce répertoire. Ces opérations sont répétées jusqu'à ce que chaque fichier ait été téléchargé. A ce moment là, la liste `pendingDirs` est vide.

Si une erreur réseau se produit lors du téléchargement du cinquième ou, disons, du vingtième fichier d'un répertoire, les fichiers restants ne sont pas téléchargés. Pour télécharger autant de fichiers que possible, une solution consisterait à planifier les opérations `GET` une par une et à attendre le signal `done(bool)` avant la planification de l'opération suivante. Dans `listInfo()`, nous accolerions simplement le nom de fichier à un `QStringList` au lieu

d'appeler `get()` directement, et dans `done(bool)` nous appellerions `get()` sur le fichier suivant à télécharger dans le `QStringList`. La séquence d'exécution serait alors celle-ci :

```
connectToHost(host, port)
login()

cd(directory_1)
list()
...
cd(directory_N)
list()
   emit listInfo(file_1_1)
   emit listInfo(file_1_2)
   ...
   emit listInfo(file_N_1)
   emit listInfo(file_N_2)
...
emit done()

get(file_1_1)
emit done()

get(file_1_2)
emit done()

...

get(file_N_1)
emit done()

get(file_N_2)
emit done()

...
```

Une autre solution consisterait à utiliser un objet `QFtp` pour chaque fichier, ce qui nous permettrait de télécharger les fichiers en parallèle, par le biais de connexions FTP séparées.

```
int main(int argc, char *argv[])
{
   QCoreApplication app(argc, argv);
   QStringList args = app.arguments();

   if (args.count() != 2) {
       cerr << "Usage: spider url" << endl
            << "Example:" << endl
            << " spider ftp://ftp.trolltech.com/freebies/leafnode"
            << endl;
       return 1;
   }
```

```
    Spider spider;
    if (!spider.getDirectory(QUrl(args[1])))
        return 1;

    QObject::connect(&spider, SIGNAL(done()), &app, SLOT(quit()));

    return app.exec();
}
```

La fonction `main()` achève le programme. Si l'utilisateur ne spécifie pas d'URL sur la ligne de commande, nous générons un message d'erreur et terminons le programme.

Dans les deux exemples FTP, les données récupérées au moyen de `get()` ont été écrites dans un `QFile`. Ceci n'est pas obligatoire. Si nous souhaitions enregistrer les données en mémoire, nous pourrions utiliser un `QBuffer`, la sous-classe `QIODevice` qui intègre un `QByteArray`. Par exemple :

```
QBuffer *buffer = new QBuffer;
buffer->open(QIODevice::WriteOnly);
ftp.get(urlInfo.name(), buffer);
```

Nous pourrions également omettre l'argument de périphérique d'E/S de `get()` ou transmettre un pointeur nul. La classe `QFtp` émet alors un signal `readyRead()` dès qu'une nouvelle donnée est disponible. Cette dernière est ensuite lue au moyen de `read()` ou de `readAll()`.

Programmer les clients HTTP

La classe QHttp implémente le côté client du protocole HTTP dans Qt. Elle fournit diverses fonctions destinées à effectuer les opérations HTTP les plus courantes dont `get()` et `post()`, et permet d'envoyer des requêtes HTTP arbitraires. Si vous avez lu la section précédente concernant `QFtp`, vous constaterez qu'il existe des similitudes entre `QFtp` et `QHttp`.

La classe `QHttp` fonctionne de façon asynchrone. Lorsque nous appelons une fonction telle que `get()` ou `post()`, elle se termine immédiatement et le transfert de données se produit ultérieurement quand le contrôle revient à la boucle d'événement de Qt. Ainsi, l'interface utilisateur de l'application reste réactive pendant le traitement des requêtes HTTP.

Nous allons étudier un exemple d'application de console nommée `httpget` qui illustre comment télécharger un fichier en utilisant le protocole HTTP. Il est très similaire à l'exemple `ftpget` de la section précédente, à la fois en matière de fonctionnalité et d'implémentation. Nous ne présenterons donc pas le fichier d'en-tête.

```
HttpGet::HttpGet(QObject *parent)
    : QObject(parent)
{
    connect(&http, SIGNAL(done(bool)), this, SLOT(httpDone(bool)));
}
```

Dans le constructeur, nous connectons le signal `done(bool)` de l'objet `QHttp` au slot privé `httpDone(bool)`.

```
bool HttpGet::getFile(const QUrl &url)
{
   if (!url.isValid()) {
       cerr << "Error: Invalid URL" << endl;
       return false;
   }

   if (url.scheme() != "http") {
       cerr << "Error: URL must start with 'http:'" << endl;
       return false;
   }

   if (url.path().isEmpty()) {
       cerr << "Error: URL has no path" << endl;
       return false;
   }

   QString localFileName = QFileInfo(url.path()).fileName();
   if (localFileName.isEmpty())
       localFileName = "httpget.out";

   file.setFileName(localFileName);
   if (!file.open(QIODevice::WriteOnly)) {
       cerr << "Error: Cannot open " << qPrintable(file.fileName())
            << " for writing: " << qPrintable(file.errorString())
            << endl;
       return false;
   }

   http.setHost(url.host(), url.port(80));
   http.get(url.path(), &file);
   http.close();
   return true;
}
```

La fonction `getFile()` effectue le même type de contrôle d'erreur que la fonction `FtpGet::getFile()` présentée précédemment et utilise la même approche pour attribuer au fichier un nom local. Lors d'une récupération depuis un site Web, aucun nom de connexion n'est nécessaire. Nous définissons simplement l'hôte et le port (en utilisant le port HTTP 80 par défaut s'il n'est pas spécifié dans l'URL) et téléchargeons les données dans le fichier, puisque le deuxième argument de `QHttp::get()` spécifie le périphérique d'E/S.

Les requêtes HTTP sont placées en file d'attente et exécutées de façon asynchrone dans la boucle d'événement de Qt. L'achèvement des requêtes est indiqué par le signal `done(bool)` de `QHttp`, que nous avons connecté à `httpDone(bool)` dans le constructeur.

```
void HttpGet::httpDone(bool error)
{
```

```
    if (error) {
        cerr << "Error: " << qPrintable(http.errorString()) << endl;
    } else {
        cerr << "File downloaded as " << qPrintable(file.fileName())
             << endl;
    }
    file.close();
    emit done();
}
```

Une fois les requêtes HTTP terminées, nous fermons le fichier, en avertissant l'utilisateur si une erreur s'est produite.

La fonction `main()` est très similaire à celle utilisée par `ftpget` :

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    QStringList args = app.arguments();

    if (args.count() != 2) {
        cerr << "Usage: httpget url" << endl
             << "Example:" << endl
             << " httpget http://doc.trolltech.com/qq/index.html"
             << endl;
        return 1;
    }

    HttpGet getter;
    if (!getter.getFile(QUrl(args[1])))
        return 1;

    QObject::connect(&getter, SIGNAL(done()), &app, SLOT(quit()));

    return app.exec();
}
```

La classe `QHttp` fournit de nombreuses opérations, dont `setHost()`, `get()`, `post()` et `head()`. Si un site requiert une authentification, `setUser()` sera utilisé pour fournir un nom d'utilisateur et un mot de passe. `QHttp` peut utiliser un socket transmis par le programmeur au lieu de son `QTcpSocket` interne, ce qui autorise l'emploi d'un `QtSslSocket` sécurisé, fourni en tant que Qt Solution par Trolltech.

Pour envoyer une liste de paires *"nom=valeur"* à un script CGI, nous faisons appel à `post()` :

```
http.setHost("www.example.com");
http.post("/cgi/somescript.py", "x=200&y=320", &file);
```

Nous pouvons transmettre les données soit sous la forme d'une chaîne de 8 octets, soit en transmettant un `QIODevice` ouvert, tel qu'un `QFile`. Pour plus de contrôle, il est possible de recourir à la fonction `request()`, qui accepte des données et un en-tête HTTP arbitraire.

Par exemple :

```
QHttpRequestHeader header("POST", "/search.html");
header.setValue("Host", "www.trolltech.com");
header.setContentType("application/x-www-form-urlencoded");
http.setHost("www.trolltech.com");
http.request(header, "qt-interest=on&search=opengl");
```

`QHttp` émet le signal `requestStarted(int)` quand il commence à exécuter une requête, puis le signal `requestFinished(int, bool)` une fois la commande terminée. Le paramètre `int` est le numéro d'ID qui identifie une requête. Si nous nous intéressons au sort des requêtes individuelles, nous pouvons stocker les numéros d'ID lors de la programmation de ces dernières. Le suivi de ces identifiants nous permet de fournir un rapport détaillé à l'utilisateur.

Dans la plupart des applications, nous souhaitons simplement savoir si la séquence de requêtes dans son ensemble s'est terminée avec succès ou non. Dans ce cas, nous établissons une connexion au signal `done()` , qui est émis lorsque la file d'attente de la requête est vide.

Quand une erreur se produit, la file d'attente de la requête est vidée automatiquement. Si nous programmons de nouvelles requêtes au moyen de l'objet `QHttp` après que l'erreur s'est produite, elles sont placées en file d'attente et envoyées.

Comme `QFtp`, `QHttp` fournit un signal `readyRead()` ainsi que les fonctions `read()` et `readAll()`, dont l'emploi évite la spécification d'un périphérique d'E/S.

Programmer les applications client/serveur TCP

Les classes `QTcpSocket` et `QTcpServer` peuvent être utilisées pour implémenter des serveurs et des clients TCP. TCP est un protocole de transport sur lequel sont basés la plupart des protocoles Internet de niveau application, y compris FTP et HTTP. En outre, il est susceptible d'être utilisé pour les protocoles personnalisés.

TCP est un protocole orienté flux. Pour les applications, les données apparaissent sous la forme d'un long flux, plutôt que sous la forme d'un gros fichier plat. Les protocoles de haut niveau basés sur TCP sont généralement orientés ligne ou bloc :

- Les protocoles orientés ligne transfèrent les données sous la forme de lignes de texte, chacune étant terminée par un retour à la ligne.
- Les protocoles orientés bloc transfèrent les donnée sous la forme de blocs de données binaires. Chaque bloc comprend un champ de taille suivi de la quantité de données spécifiée.

`QTcpSocket` hérite de `QIODevice` par le biais de `QAbstractSocket`. Il peut donc être lu et écrit au moyen d'un `QDataStream` ou d'un `QTextStream`. Une différence notable entre la lecture de données à partir d'un réseau et celle effectuée depuis un fichier est que nous devons veiller à avoir reçu suffisamment de données avant d'utiliser l'opérateur >>. Dans le cas contraire, nous obtenons un comportement aléatoire.

Dans cette section, nous allons examiner le code d'un client et d'un serveur qui utilisent un protocole personnalisé orienté bloc. Le client se nomme Trip Planner et permet aux utilisateurs de planifier leur prochain voyage ferroviaire. Le serveur se nomme Trip Server et fournit les informations concernant le voyage au client. Nous allons commencer par écrire le client Trip Planner.

Trip Planner fournit les champs `From`, `To`, `Date` et `Approximate Time` ainsi que deux boutons d'option indiquant si l'heure approximative est celle de départ ou d'arrivée. Lorsque l'utilisateur clique sur `Search`, l'application expédie une requête au serveur, qui renvoie une liste des trajets correspondant aux critères de l'utilisateur. La liste est présentée sous la forme d'un `QTableWidget` dans la fenêtre Trip Planner. Le bas de la fenêtre est occupé par un `QProgressBar` ainsi que par un `QLabel` qui affiche le statut de la dernière opération. (Voir Figure 14.1)

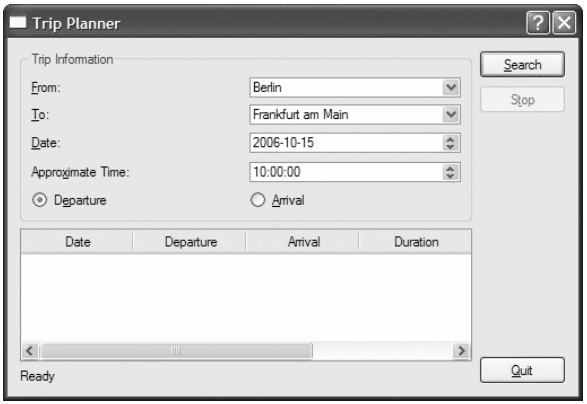


Figure 14.1
L'application Trip Planner

L'interface utilisateur de Trip Planner a été créée au moyen de *Qt Designer* dans un fichier nommé `tripplanner.ui`. Ici, nous allons nous concentrer sur le code source de la sous-classe `QDialog` qui implémente la fonctionnalité de l'application :

```
#include "ui_tripplanner.h"

class TripPlanner : public QDialog, public Ui::TripPlanner
{
   Q_OBJECT

public:
   TripPlanner(QWidget *parent = 0);

private slots:
   void connectToServer();
   void sendRequest();
   void updateTableWidget();
   void stopSearch();
   void connectionClosedByServer();
   void error();
```

```
private:
   void closeConnection();

   QTcpSocket tcpSocket;
   quint16 nextBlockSize;
};
```

La classe `TripPlanner` hérite de `Ui::TripPlanner` (qui est généré par le `uic` de `tripplanner.ui`) en plus de `QDialog`. La variable membre `tcpSocket` encapsule la connexion TCP. La variable `nextBlockSize` est utilisée lors de l'analyse des blocs reçus du serveur.

```
TripPlanner::TripPlanner(QWidget *parent)
   : QDialog(parent)
{
   setupUi(this);
   QDateTime dateTime = QDateTime::currentDateTime();
   dateEdit->setDate(dateTime.date());
   timeEdit->setTime(QTime(dateTime.time().hour(), 0));

   progressBar->hide();
   progressBar->setSizePolicy(QSizePolicy::Preferred,
                              QSizePolicy::Ignored);

   tableWidget->verticalHeader()->hide();
   tableWidget->setEditTriggers(QAbstractItemView::NoEditTriggers);

   connect(searchButton, SIGNAL(clicked()),
           this, SLOT(connectToServer()));
   connect(stopButton, SIGNAL(clicked()), this, SLOT(stopSearch()));

   connect(&tcpSocket, SIGNAL(connected()), this, SLOT(sendRequest()));
   connect(&tcpSocket, SIGNAL(disconnected()),
           this, SLOT(connectionClosedByServer()));
   connect(&tcpSocket, SIGNAL(readyRead()),
           this, SLOT(updateTableWidget()));
   connect(&tcpSocket, SIGNAL(error(QAbstractSocket::SocketError)),
           this, SLOT(error()));
}
```

Dans le constructeur, nous initialisons les éditeurs de date et d'heure en fonction de la date et de l'heure courantes. Nous masquons également la barre de progression, car nous souhaitons ne l'afficher que lorsqu'une connexion est active. Dans *Qt Designer*, les propriétés `minimum` et `maximum` de la barre de progression sont toutes deux définies en `0`, ce qui indique au `QProgressBar` de se comporter comme un indicateur d'activité au lieu d'une barre de progression standard basée sur les pourcentages.

En outre, dans le constructeur, nous connectons les signaux `connected()`, `disconnected()`, `readyRead()` et `error(QAbstractSocket::SocketError)` de `QTcpSocket` à des slots privés.

```
void TripPlanner::connectToServer()
{
   tcpSocket.connectToHost("tripserver.zugbahn.de", 6178);

   tableWidget->setRowCount(0);
   searchButton->setEnabled(false);
   stopButton->setEnabled(true);
   statusLabel->setText(tr("Connecting to server..."));
   progressBar->show();

   nextBlockSize = 0;
}
```

Le slot `connectToServer()` est exécuté lorsque l'utilisateur clique sur `Search` pour lancer une recherche. Nous appelons `connectToHost()` sur l'objet `QTcpSocket` pour établir une connexion avec le serveur, que nous supposons être accessible par le biais du port 6178 sur l'hôte fictif `tripserver.zugbahn.de`. (Si vous souhaitez tester l'exemple sur votre propre machine, remplacez le nom de l'hôte par `QHostAddress::LocalHost`.) L'appel de `connectToHost()` est asynchrone. Il rend toujours le contrôle immédiatement. La connexion est généralement établie ultérieurement. L'objet `QTcpSocket` émet le signal `connected()` lorsque la connexion fonctionne ou `error(QAbstractSocket::SocketError)` en cas d'échec.

Nous mettons ensuite à jour l'interface utilisateur, en particulier en affichant la barre de progression.

Nous définissons enfin la variable `nextBlockSize` en `0`. Cette variable stocke la longueur du prochain bloc en provenance du serveur. Nous avons choisi d'utiliser la valeur `0` pour indiquer que nous ne connaissons pas encore la taille du bloc à venir.

```
void TripPlanner::sendRequest()
{
   QByteArray block;
   QDataStream out(&block, QIODevice::WriteOnly);
   out.setVersion(QDataStream::Qt_4_1);
   out << quint16(0) << quint8('S') << fromComboBox->currentText()
       << toComboBox->currentText() << dateEdit->date()
       << timeEdit->time();

   if (departureRadioButton->isChecked()) {
       out << quint8('D');
   } else {
       out << quint8('A');
   }
   out.device()->seek(0);
   out << quint16(block.size() - sizeof(quint16));
   tcpSocket.write(block);

   statusLabel->setText(tr("Sending request..."));
}
```

Le slot `sendRequest()` est exécuté lorsque l'objet `QTcpSocket` émet le signal `connected()`, indiquant qu'une connexion a été établie. La tâche du slot consiste à générer une requête à destination du serveur, avec toutes les informations entrées par l'utilisateur.

La requête est un bloc binaire au format suivant :

`quint16`	Taille du bloc en octets (excluant ce champ)
`quint8`	Type de requête (toujours 'S')
`QString`	Ville de départ
`QString`	Ville d'arrivée
`QDate`	Date du voyage
`QTime`	Horaire approximatif du voyage
`quint8`	Heure de départ ('D') ou d'arrivée ('A')

Dans un premier temps, nous écrivons les données dans un `QByteArray` nommé `block`. Nous ne pouvons pas écrire les données directement dans le `QTcpSocket` car nous ne connaissons pas la taille du bloc avant d'y avoir placé toutes les données.

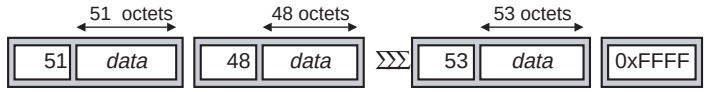
Nous avons initialement indiqué `0` pour la taille du bloc, suivi du reste des données. Nous appelons ensuite `seek(0)` sur le périphérique d'E/S (un `QBuffer` créé par `QDataStream` à l'arrière-plan) pour revenir au début du tableau d'octets, et nous remplaçons le `0` initial par la taille des données du bloc. Elle est calculée en prenant la taille du bloc et en soustrayant `sizeof(quint16)` (c'est-à-dire 2) pour exclure le champ de taille du compte d'octets. Nous appelons alors `write()` sur le `QTcpSocket` pour envoyer le bloc au serveur.

```
void TripPlanner::updateTableWidget()
{
    QDataStream in(&tcpSocket);
    in.setVersion(QDataStream::Qt_4_1);

    forever {
        int row = tableWidget->rowCount();

        if (nextBlockSize == 0) {
            if (tcpSocket.bytesAvailable() < sizeof(quint16))
                break;
            in >> nextBlockSize;
        }

        if (nextBlockSize == 0xFFFF) {
            closeConnection();
            statusLabel->setText(tr("Found %1 trip(s)").arg(row));
            break;
        }
```

```
        if (tcpSocket.bytesAvailable() < nextBlockSize)
            break;

        QDate date;
        QTime departureTime;
        QTime arrivalTime;
        quint16 duration;
        quint8 changes;
        QString trainType;

        in >> date >> departureTime >> duration >> changes >> trainType;
        arrivalTime = departureTime.addSecs(duration * 60);

        tableWidget->setRowCount(row + 1);

        QStringList fields;
        fields << date.toString(Qt::LocalDate)
               << departureTime.toString(tr("hh:mm"))
               << arrivalTime.toString(tr("hh:mm"))
               << tr("%1 hr %2 min").arg(duration / 60)
                                    .arg(duration % 60)
               << QString::number(changes)
               << trainType;
        for (int i = 0; i < fields.count(); ++i)
            tableWidget->setItem(row, i,
                                 new QTableWidgetItem(fields[i]));
        nextBlockSize = 0;
    }
}
```

Le slot `updateTableWidget()` est connecté au signal `readyRead()` de `QTcpSocket`, qui est émis dès que le `QTcpSocket` reçoit de nouvelles données en provenance du serveur. Le serveur nous envoie une liste des trajets possibles correspondant aux critères de l'utilisateur. Chaque trajet est expédié sous la forme d'un bloc unique. La boucle `forever` est nécessaire dans la mesure où nous ne recevons pas obligatoirement un seul bloc de données à la fois de la part du serveur. Nous pouvons recevoir un bloc entier, une partie de celui-ci, un bloc et demi ou encore tous les blocs à la fois. (Voir Figure 14.2)

Figure 14.2
Les blocs de Trip Server



Comment fonctionne la boucle `forever` ? Si la variable `nextBlockSize` a pour valeur `0`, nous n'avons pas lu la taille du bloc suivant. Nous essayons de la lire (en prenant en compte le fait que deux octets au moins sont disponibles pour la lecture). Le serveur utilise une valeur de taille de `0xFFFF` pour indiquer qu'il ne reste plus de données à recevoir. Ainsi, si nous lisons cette valeur, nous savons que nous avons atteint la fin.

Si la taille du bloc n'est pas de `0xFFFF`, nous essayons de lire le bloc suivant. Dans un premier temps, nous essayons de déterminer si des octets de taille de bloc sont disponibles à la lecture. Si tel n'est pas le cas, nous interrompons cette action un instant. Le signal `readyRead()` sera de nouveau émis lorsque des données supplémentaires seront disponibles. Nous procéderons alors à de nouvelles tentatives.

Lorsque nous sommes certains que le bloc entier est arrivé, nous pouvons utiliser l'opérateur `>>` en toute sécurité sur le `QDataStream` pour extraire les informations relatives au voyage, et nous créons un `QTableWidgetItems` avec ces informations. Le format d'un bloc reçu du serveur est le suivant :

`quint16`	Taille du bloc en octets (en excluant son champ)
`QDate`	Date de départ
`QTime`	Heure de départ
`quint16`	Durée (en minutes)
`quint8`	Nombre de changements
`QString`	Type de train

A la fin, nous réinitialisons la variable `nextBlockSize` en `0` pour indiquer que la taille du bloc suivant est inconnue et doit être lue.

```
void TripPlanner::closeConnection()
{
   tcpSocket.close();
   searchButton->setEnabled(true);
   stopButton->setEnabled(false);
   progressBar->hide();
}
```

La fonction privée `closeConnection()` ferme la connexion avec le serveur TCP et met à jour l'interface utilisateur. Elle est appelée depuis `updateTableWidget()` lorsque `0xFFFF` est lu ainsi que depuis plusieurs autres slots sur lesquels nous reviendrons dans un instant.

```
void TripPlanner::stopSearch()

{
   statusLabel->setText(tr("Search stopped"));
   closeConnection();
}
```

Le slot `stopSearch()` est connecté au signal `clicked()` du bouton `Stop`. Il appelle simplement `closeConnection()`.

```
void TripPlanner::connectionClosedByServer()
{
```

```
    if (nextBlockSize != 0xFFFF)
        statusLabel->setText(tr("Error: Connection closed by server"));
    closeConnection();
}
```

Le slot `connectionClosedByServer()` est connecté au signal `disconnected()` du `QTcpSocket`. Si le serveur ferme la connexion sans que nous ayons encore reçu le marqueur `0xFFFF` de fin de données, nous indiquons à l'utilisateur qu'une erreur s'est produite. Nous appelons normalement `closeConnection()` pour mettre à jour l'interface utilisateur.

```
void TripPlanner::error()
{
    statusLabel->setText(tcpSocket.errorString());
    closeConnection();
}
```

Le slot `error()` est connecté au signal `error(QAbstractSocket::SocketError)` du `QTcpSocket`. Nous ignorons le code d'erreur et nous utilisons `QTcpSocket::errorString()`, qui retourne un message en texte clair concernant la dernière erreur détectée.

Tout ceci concerne la classe `TripPlanner`. Comme nous pouvons nous y attendre, la fonction `main()` de l'application TripPlanner est la suivante :

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    TripPlanner tripPlanner;
    tripPlanner.show();
    return app.exec();
}
```

Implémentons maintenant le serveur. Ce dernier est composé de deux classes : `TripServer` et `ClientSocket`. La classe `TripServer` hérite de `QTcpServer`, une classe qui nous permet d'accepter des connexions TCP entrantes. `ClientSocket` réimplémente `QTcpSocket` et gère une connexion unique. Il existe à chaque instant autant d'objets `ClientSocket` en mémoire que de clients servis.

```
class TripServer : public QTcpServer
{
    Q_OBJECT

public:
    TripServer(QObject *parent = 0);

private:
    void incomingConnection(int socketId);
};
```

La classe `TripServer` réimplémente la fonction `incomingConnection()` depuis `QTcpServer`. Cette fonction est appelée dès qu'un client tente d'établir une connexion au port écouté par le serveur.

```
TripServer::TripServer(QObject *parent)
   : QTcpServer(parent)
{
}
```

Le constructeur `TripServer` est simple.

```
void TripServer::incomingConnection(int socketId)
{
   ClientSocket *socket = new ClientSocket(this);
   socket->setSocketDescriptor(socketId);
}
```

Dans `incomingConnection()`, nous créons un objet `ClientSocket` qui est un enfant de l'objet `TripServer`, et nous attribuons à son descripteur de socket le nombre qui nous a été fourni. L'objet `ClientSocket` se supprimera automatiquement de lui-même une fois la connexion terminée.

```
class ClientSocket : public QTcpSocket
{
   Q_OBJECT

public:
   ClientSocket(QObject *parent = 0);

private slots:
   void readClient();

private:
   void generateRandomTrip(const QString &from, const QString &to,
                           const QDate &date, const QTime &time);

   quint16 nextBlockSize;
};
```

La classe `ClientSocket` hérite de `QTcpSocket` et encapsule l'état d'un client unique.

```
ClientSocket::ClientSocket(QObject *parent)
   : QTcpSocket(parent)
{
   connect(this, SIGNAL(readyRead()), this, SLOT(readClient()));
   connect(this, SIGNAL(disconnected()), this, SLOT(deleteLater()));
   nextBlockSize = 0;
}
```

Dans le constructeur, nous établissons les connexions signal/slot nécessaires, et nous définissons la variable `nextBlockSize` en `0`, indiquant ainsi que nous ne connaissons pas encore la taille du bloc envoyé par le client.

Le signal `disconnected()` est connecté à `deleteLater()`, une fonction héritée de `QObject` qui supprime l'objet lorsque le contrôle retourne à la boucle d'événement de Qt. De cette façon, l'objet `ClientSocket` est supprimé lorsque la connexion du socket est fermée.

```
void ClientSocket::readClient()
{
   QDataStream in(this);
   in.setVersion(QDataStream::Qt_4_1);

   if (nextBlockSize == 0) {
      if (bytesAvailable() < sizeof(quint16))
         return;
      in >> nextBlockSize;
   }
   if (bytesAvailable() < nextBlockSize)
      return;

   quint8 requestType;
   QString from;
   QString to;
   QDate date;
   QTime time;
   quint8 flag;

   in >> requestType;
   if (requestType == 'S') {
      in >> from >> to >> date >> time >> flag;

      srand(from.length() * 3600 + to.length() * 60 + time.hour());
      int numTrips = rand() % 8;
      for (int i = 0; i < numTrips; ++i)
         generateRandomTrip(from, to, date, time);

      QDataStream out(this);
      out << quint16(0xFFFF);
   }

   close();
}
```

Le slot `readClient()` est connecté au signal `readyRead()` du `QTcpSocket`. Si `nextBlockSize` est défini en `0`, nous commençons par lire la taille du bloc. Dans le cas contraire, nous l'avons déjà lue. Nous poursuivons donc en vérifiant si un bloc entier est arrivé, et nous le lisons d'une seule traite. Nous utilisons `QDataStream` directement sur le `QTcpSocket` (l'objet `this`) et lisons les champs au moyen de l'opérateur `>>`.

Une fois la requête du client lue, nous sommes prêts à générer une réponse. Dans le cas d'une application réelle, nous rechercherions les informations dans une base de données d'horaires et tenterions de trouver les trajets correspondants. Ici, nous nous contenterons d'une fonction nommée `generateRandomTrip()` qui générera un trajet aléatoire. Nous appelons la fonction un nombre quelconque de fois, puis nous envoyons `0xFFFF` pour signaler la fin des données. Enfin, nous fermons la connexion.

```
void ClientSocket::generateRandomTrip(const QString & /* from */,
      const QString & /* to */, const QDate &date, const QTime &time)
```

```
{
    QByteArray block;
    QDataStream out(&block, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_1);
    quint16 duration = rand() % 200;
    out << quint16(0) << date << time << duration << quint8(1)
        << QString("InterCity");
    out.device()->seek(0);
    out << quint16(block.size() - sizeof(quint16));

    write(block);
}
```

La fonction `generateRandomTrip()` illustre comment envoyer un bloc de données par le biais d'une connexion TCP. Ce processus est très similaire à celui suivi sur le client, dans la fonction `sendRequest()`. Une fois encore, nous écrivons le bloc dans un `QByteArray` en exécutant `write()`, de façon à pouvoir déterminer sa taille avant de l'envoyer.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    TripServer server;
    if (!server.listen(QHostAddress::Any, 6178)) {
        cerr << "Failed to bind to port" << endl;
        return 1;
    }

    QPushButton quitButton(QObject::tr("&Quit"));
    quitButton.setWindowTitle(QObject::tr("Trip Server"));
    QObject::connect(&quitButton, SIGNAL(clicked()),
                     &app, SLOT(quit()));
    quitButton.show();
    return app.exec();
}
```

Dans `main()`, nous créons un objet `TripServer` et un `QPushButton` qui permet à l'utilisateur d'arrêter le serveur. Nous lançons le serveur en appelant `QTcpSocket::listen()`, qui reçoit l'adresse IP et le numéro de port sur lequel nous souhaitons accepter les connexions. L'adresse spéciale `0.0.0.0` (`QHostAddress::Any`) signifie toute interface IP présente sur l'hôte local.

Ceci termine notre exemple client/serveur. Ici, nous avons utilisé un protocole orienté bloc qui nous permet de faire appel à `QDataStream` pour la lecture et l'écriture. Si nous souhaitions utiliser un protocole orienté ligne, l'approche la plus simple serait de recourir aux fonctions `canReadLine()` et `readLine()` de `QTcpSocket` dans un slot connecté au signal `readyRead()` :

```
QStringList lines;
while (tcpSocket.canReadLine())
    lines.append(tcpSocket.readLine());
```

Nous traiterions alors chaque ligne lue. Comme pour l'envoi des données, ceci pourrait être effectué en utilisant un `QTextStream` sur le `QTcpSocket`.

L'implémentation serveur que nous avons utilisée n'est pas adaptée à une situation où les connexions sont nombreuses. En effet, lorsque nous traitons une requête, nous ne gérons pas les autres connexions. Une approche plus souple consisterait à démarrer un nouveau thread pour chaque connexion. L'exemple Threaded Fortune Server situé dans le répertoire `examples/network/threadedfortuneserver` illustre ce procédé.

Envoi et réception de datagrammes UDP

La classe `QUdpSocket` peut être utilisée pour envoyer et recevoir des datagrammes UDP. UDP est un protocole orienté datagramme non fiable. Certains protocoles de niveau application utilisent UDP car il est plus léger que TCP. Avec UDP, les données sont envoyées sous forme de paquets (datagrammes) d'un hôte à un autre. Il n'existe pas de concept de connexion, et si un paquet UDP n'est pas remis avec succès, aucune erreur n'est signalée à l'expéditeur. (Voir Figure 14.3)

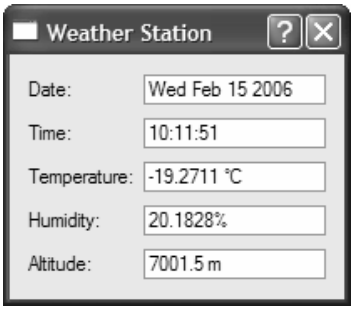


Figure 14.3
L'application Weather Station

Les exemples Weather Balloon et Weather Station vous montreront comment utiliser UDP à partir d'une application Qt. L'application Weather Balloon reproduit un ballon météo qui envoie un datagramme UDP (au moyen d'une connexion sans fil) contenant les conditions atmosphériques courantes toutes les deux secondes. L'application Weather Station reçoit ces datagrammes et les affiche à l'écran. Nous allons commencer par le code du Weather Ballon.

```
class WeatherBalloon : public QPushButton
{
   Q_OBJECT
public:
   WeatherBalloon(QWidget *parent = 0);

   double temperature() const;
   double humidity() const;
```

```
    double altitude() const;

private slots:
    void sendDatagram();

private:
    QUdpSocket udpSocket;
    QTimer timer;
};
```

La classe `WeatherBalloon` hérite de `QPushButton`. Elle utilise sa variable privée `QUdpSocket` pour communiquer avec la station météo (Weather Station).

```
WeatherBalloon::WeatherBalloon(QWidget *parent)
    : QPushButton(tr("Quit"), parent)
{
    connect(this, SIGNAL(clicked()), this, SLOT(close()));
    connect(&timer, SIGNAL(timeout()), this, SLOT(sendDatagram()));

    timer.start(2 * 1000);

    setWindowTitle(tr("Weather Balloon"));
}
```

Dans le constructeur, nous lançons un `QTimer` pour invoquer `sendDatagram()` toutes les deux secondes.

```
void WeatherBalloon::sendDatagram()
{
    QByteArray datagram;
    QDataStream out(&datagram, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_1);
    out << QDateTime::currentDateTime() << temperature() << humidity()
        << altitude();

    udpSocket.writeDatagram(datagram, QHostAddress::LocalHost, 5824);
}
```

Dans `sendDatagram()`, nous générons et envoyons un datagramme contenant la date, l'heure, la température, l'humidité et l'altitude :

`QDateTime`	Date et heure de mesure
`double`	Température (en ˚C)
`double`	Humidité (en %)
`double`	Altitude (in mètres)

Le datagramme est expédié au moyen de `QUdpSocket::writeDatagram()`. Les deuxième et troisième arguments de `writeDatagram()` sont l'adresse IP et le numéro de port de l'homologue (la Weather Station). Nous supposons ici que la Weather Station s'exécute sur la même machine que le Weather Balloon. Nous utilisons donc l'adresse IP 127.0.0.1 (`QHostAddress::LocalHost`), une adresse spéciale qui désigne l'hôte local.

Contrairement aux sous-classes de `QAbstractSocket`, `QUdpSocket` accepte uniquement les adresses d'hôte, mais pas les noms. Si nous devions convertir un nom d'hôte en son adresse IP, deux solutions s'offriraient à nous : si nous nous sommes préparés à un blocage pendant la recherche, nous pouvons faire appel à la fonction statique `QHostInfo::fromName()`. Dans le cas contraire, nous employons la fonction statique `QHostInfo::lookupHost()`, qui rend le contrôle immédiatement et, une fois la recherche terminée, appelle le slot qui lui est transmis avec un objet `QHostInfo` contenant les adresses correspondantes.

```
int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   WeatherBalloon balloon;
   balloon.show();
   return app.exec();
}
```

La fonction `main()` crée simplement un objet `WeatherBalloon`, qui sert à la fois d'homologue UDP et de `QPushButton` à l'écran. En cliquant sur le `QPushButton`, l'utilisateur quitte l'application.

Revenons maintenant au code source du client Weather Station.

```
class WeatherStation : public QDialog
{
   Q_OBJECT

public:
   WeatherStation(QWidget *parent = 0);

private slots:
   void processPendingDatagrams();

private:
   QUdpSocket udpSocket;

   QLabel *dateLabel;
   QLabel *timeLabel;
   ...
   QLineEdit *altitudeLineEdit;
};
```

La classe `WeatherStation` hérite de `QDialog`. Elle écoute un port UDP particulier, analyse tous les datagrammes entrants (en provenance du Weather Balloon) et affiche leur contenu dans cinq `QLineEdits` en lecture seulement. La seule variable privée présentant un intérêt ici

est la variable `udpSocket` du type `QUdpSocket`, à laquelle nous allons faire appel pour recevoir les datagrammes.

```
WeatherStation::WeatherStation(QWidget *parent)
   : QDialog(parent)
{
   udpSocket.bind(5824);

   connect(&udpSocket, SIGNAL(readyRead()),
      this, SLOT(processPendingDatagrams()));
   ...
}
```

Dans le constructeur, nous commençons par établir une liaison entre le `QUdpSocket` et le port auquel le Weather Balloon transmet ses données. Comme nous n'avons pas spécifié d'adresse hôte, le socket accepte les datagrammes envoyés à n'importe quelle adresse IP appartenant à la machine sur laquelle s'exécute la Weather Station. Puis nous connectons le signal `readyRead()` du socket au `processPendingDatagrams()` privé qui extrait les données et les affiche.

```
void WeatherStation::processPendingDatagrams()
{
   QByteArray datagram;

   do {
      datagram.resize(udpSocket.pendingDatagramSize());
      udpSocket.readDatagram(datagram.data(), datagram.size());
   } while (udpSocket.hasPendingDatagrams());

   QDateTime dateTime;
   double temperature;
   double humidity;
   double altitude;

   QDataStream in(&datagram, QIODevice::ReadOnly);
   in.setVersion(QDataStream::Qt_4_1);
   in >> dateTime >> temperature >> humidity >> altitude;

   dateLineEdit->setText(dateTime.date().toString());
   timeLineEdit->setText(dateTime.time().toString());
   temperatureLineEdit->setText(tr("%1 °C").arg(temperature));
   humidityLineEdit->setText(tr("%1%").arg(humidity));
   altitudeLineEdit->setText(tr("%1 m").arg(altitude));
}
```

Le slot `processPendingDatagrams()` est appelé quand un datagramme est arrivé. `QUdpSocket` place les datagrammes entrants en file d'attente et nous permet d'y accéder un par un. Normalement, il ne devrait y avoir qu'un seul datagramme, mais nous ne pouvons pas exclure la possibilité que l'expéditeur en envoie plusieurs à la fois avant que le signal `readyRead()` ne soit émis. Dans ce cas, nous les ignorons tous, à l'exception du dernier. Les précédents véhiculent en effet des informations obsolètes.

La fonction `pendingDatagramSize()` retourne la taille du premier datagramme en attente. Du point de vue de l'application, les datagrammes sont toujours envoyés et reçus sous la forme d'une unité de données unique. Ainsi, si des octets quelconques sont disponibles, le datagramme entier peut être lu. L'appel de `readDatagram()` copie le contenu du premier datagramme en attente dans la mémoire tampon `char*` spécifiée (en troquant les données si la capacité de cette mémoire n'est pas suffisante) et passe au datagramme suivant en attente. Une fois tous les datagrammes lus, nous décomposons le dernier (celui avec les mesures atmosphériques les plus récentes) et alimentons le `QLineEdits` avec les nouvelles données.

```
int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   WeatherStation station;
   station.show();
   return app.exec();
}
```

Enfin, nous créons et affichons la `WeatherStation` dans `main()`.

Nous en avons maintenant terminé avec notre émetteur et destinataire UDP. Les applications sont aussi simples que possible, puisque le Weather Balloon envoie des datagrammes à la Weather Station qui les reçoit. Dans la plupart des cas du monde réel, les deux applications auraient besoin d'effectuer des opérations de lecture et d'écriture sur leur socket.

Un numéro de port et une adresse hôte peuvent être transmis aux fonctions `QUdpSocket::writeDatagram()`, de sorte que le `QUdpSocket` puisse réaliser une lecture depuis l'hôte et le port auquel il est lié avec `bind()`, et effectuer une opération de lecture vers un autre hôte ou port.

15

XML

Au sommaire de ce chapitre

- ✔ Lire du code XML avec SAX
- ✔ Lire du code XML avec DOM
- ✔ Ecrire du code XML

XML (*Extensible Markup Language*) est un format de fichier texte polyvalent, populaire pour l'échange et le stockage des données. Qt fournit deux API distinctes faisant partie du module *QtXml* pour la lecture de documents XML :

- SAX (*Simple API for XML*) rapporte des "événements d'analyse" directement à l'application par le biais de fonctions virtuelles.
- DOM (*Document Object Model*) convertit une documentation XML en une structure arborescente, que l'application peut parcourir.

Trois facteurs principaux sont à prendre en compte lors du choix entre DOM et SAX pour une application particulière. SAX est de niveau inférieur et généralement plus rapide, ce qui le rend particulièrement approprié pour des tâches simples (telles que la recherche de toutes les occurrences d'une balise donnée dans un document XML) ou

pour la lecture de fichiers de très grande taille pour lesquels la mémoire sera insuffisante. Mais pour de nombreuses applications, la commodité de DOM prime sur la vitesse potentielle et les avantages offerts par SAX concernant la mémoire.

Pour écrire des fichiers XML, deux options sont disponibles : nous pouvons générer le code XML manuellement, ou représenter les données sous la forme d'un arbre DOM en mémoire et demander à ce dernier de s'écrire par lui-même dans un fichier.

Lire du code XML avec SAX

SAX est une API standard de domaine public destinée à la lecture de documents XML. Les classes SAX de Qt sont modelées sur l'implémentation SAX2 de Java, avec quelques différences dans l'attribution des noms afin de s'adapter aux conventions de Qt. Pour plus d'informations concernant SAX, reportez-vous à l'adresse **http://www.saxproject.org/**.

Qt fournit un analyseur XML basé sur SAX nommé `QXmlSimpleReader`. Cet analyseur reconnaît du code XML bien formé et prend en charge les espaces de noms XML. Quand il parcourt le document, il appelle les fonctions virtuelles des classes gestionnaires enregistrées pour signaler des événements d'analyse. (Ces "événements d'analyse" n'ont aucun rapport avec les événements Qt, tels que les événements touche et souris.) Supposons, par exemple que l'analyseur examine le document XML suivant :

```
<doc>
   <quote>Ars longa vita brevis</quote>
</doc>
```

Il appelle les gestionnaires d'événements d'analyse ci-après :

```
startDocument()
startElement("doc")
startElement("quote")
characters("Ars longa vita brevis")
endElement("quote")
endElement("doc")
endDocument()
```

Ces fonctions sont toutes déclarées dans `QXmlContentHandler`. Pour des questions de simplicité, nous avons omis certains arguments de `startElement()` et `endElement()`.

`QXmlContentHandler` est juste l'une des nombreuses classes gestionnaires susceptible d'être utilisée avec `QXmlSimpleReader`. Les autres sont `QXmlEntityResolver`, `QXmlDTDHandler`, `QXmlErrorHandler`, `QXmlDeclHandler` et `QXmlLexicalHandler`. Ces classes ne déclarent que des fonctions purement virtuelles et fournissent des informations concernant les différents types d'événements d'analyse. Pour la plupart des applications, `QXmlContentHandler` et `QXmlErrorHandler` sont les deux seules nécessaires.

Pour des raisons de commodité, Qt fournit aussi `QXmlDefaultHandler`, une classe qui hérite de toutes les classes gestionnaires et qui fournit des implémentations simples de toutes les fonctions. Une telle conception, avec de nombreuses classes gestionnaires abstraites et une sous-classe simple, est inhabituelle pour Qt. Elle a été adoptée pour se conformer à l'implémentation de modèle Java.

Nous allons étudier un exemple qui illustre comment utiliser `QXmlSimpleReader` et `QXmlDefaultHandler` pour analyser un fichier XML ad hoc et afficher son contenu dans un `QTreeWidget`. La sous-classe `QXmlDefaultHandler` se nomme `SaxHandler`, et le format géré par celle-ci est celui d'un index de livre, avec les entrées et les sous-entrées.

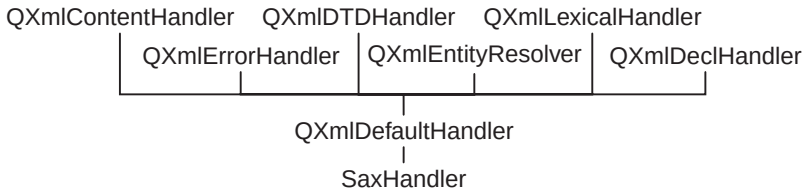


Figure 15.1
Arbre d'héritage pour `SaxHandler`

Voici le fichier d'index qui est affiché dans le `QTreeWidget` en Figure 15.2 :

```
<?xml version="1.0"?>
<bookindex>
   <entry term="sidebearings">
      <page>10</page>
      <page>34-35</page>
      <page>307-308</page>
   </entry>
   <entry term="subtraction">
      <entry term="of pictures">
         <page>115</page>
         <page>244</page>
      </entry>
      <entry term="of vectors">
         <page>9</page>
      </entry>
   </entry>
</bookindex>
```

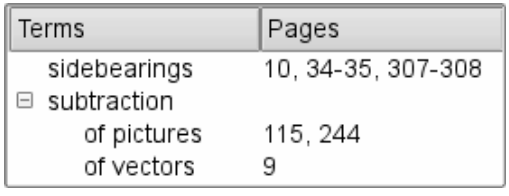


Figure 15.2
Un fichier d'index affiché dans un `QTreeWidget`

La première étape dans l'implémentation de l'analyseur consiste à définir la sous-classe `QXmlDefaultHandler` :

```
class SaxHandler : public QXmlDefaultHandler
{
public:
   SaxHandler(QTreeWidget *tree);

   bool startElement(const QString &namespaceURI,
                     const QString &localName,
                     const QString &qName,
                     const QXmlAttributes &attributes);
   bool endElement(const QString &namespaceURI,
                   const QString &localName,
                   const QString &qName);
   bool characters(const QString &str);
   bool fatalError(const QXmlParseException &exception);

private:
   QTreeWidget *treeWidget;
   QTreeWidgetItem *currentItem;
   QString currentText;
};
```

La classe `SaxHandler` hérite de `QXmlDefaultHandler` et réimplémente quatre fonctions : `startElement()`, `endElement()`, `characters()` et `fatalError()`. Les trois premières sont déclarées dans `QXmlContentHandler`. La dernière est déclarée dans `QXmlErrorHandler`.

```
SaxHandler::SaxHandler(QTreeWidget *tree)
{
   treeWidget = tree;
   currentItem = 0;
}
```

Le constructeur `SaxHandler` accepte le `QTreeWidget` que nous souhaitons remplir avec les informations stockées dans le fichier XML.

```
bool SaxHandler::startElement(const QString & /* namespaceURI */,
                              const QString & /* localName */,
                              const QString &qName,
                              const QXmlAttributes &attributes)
{
   if (qName == "entry") {
      if (currentItem) {
          currentItem = new QTreeWidgetItem(currentItem);
      } else {
          currentItem = new QTreeWidgetItem(treeWidget);
      }
```

```
            currentItem->setText(0, attributes.value("term"));
        } else if (qName == "page") {
            currentText.clear();
        }
        return true;
    }
```

La fonction `startElement()` est appelée dès que le lecteur rencontre une nouvelle balise d'ouverture. Le troisième paramètre est le nom de la balise (ou plus précisément, son "nom qualifié"). Le quatrième paramètre est la liste des attributs. Dans cet exemple, nous ignorons les premier et deuxième paramètres. Ils sont utiles pour les fichiers XML qui utilisent le mécanisme d'espace de noms de XML, un sujet qui est traité en détail dans la documentation de référence.

Si la balise est `<entry>`, nous créons un nouvel élément `QTreeWidget`. Si elle est imbriquée dans une autre balise `<entry>`, la nouvelle balise définit une sous-entrée dans l'index, et le nouveau `QTreeWidgetItem` est créé en tant qu'enfant du `QTreeWidgetItem` qui représente l'entrée principale. Dans le cas contraire, nous créons le `QTreeWidgetItem` avec l'élément `treeWidget` en tant que parent, en faisant de celui-ci un élément de haut niveau. Nous appelons `setText()` pour définir le texte présenté en colonne 0 avec la valeur de l'attribut `term` de la balise `<entry>`.

Si la balise est `<page>`, nous définissons le `currentText` en une chaîne vide. Le `currentText` sert d'accumulateur pour le texte situé entre les balises `<page>` et `</page>`.

Nous retournons enfin `true` pour demander à SAX de poursuivre l'analyse du fichier. Si nous souhaitions signaler les balises inconnues comme des erreurs, nous retournerions `false` dans ces situations. Nous réimplémenterions également `errorString()` à partir de `QXmlDefaultHandler` pour retourner un message d'erreur approprié.

```
bool SaxHandler::characters(const QString &str)
{
    currentText += str;
    return true;
}
```

La fonction `characters()` est appelée si des données caractères sont rencontrées dans le document XML. Nous accolons simplement les caractères à la variable `currentText`.

```
bool SaxHandler::endElement(const QString & /* namespaceURI */,
                            const QString & /* localName */,
                            const QString &qName)
{
    if (qName == "entry") {
        currentItem = currentItem->parent();
    } else if (qName == "page") {
        if (currentItem) {
            QString allPages = currentItem->text(1);
```

```
            if (!allPages.isEmpty())
                allPages += ", ";
            allPages += currentText;
            currentItem->setText(1, allPages);
        }
    }
    return true;
}
```

La fonction `endElement()` est appelée quand le lecteur rencontre une balise de fermeture. Comme pour `startElement()`, le troisième paramètre est le nom de la balise.

Si la balise est `</entry>`, nous mettons à jour la variable privée `currentItem` de façon à la diriger vers le parent de `QTreeWidgetItem` en cours. De cette façon, la variable `currentItem` reprend la valeur qui était la sienne avant la lecture de la balise `<entry>` correspondante.

Si la balise est `</page>`, nous ajoutons le numéro de page ou la plage de pages sous la forme d'une liste séparée par des virgules au texte de l'élément courant de la colonne 1.

```
bool SaxHandler::fatalError(const QXmlParseException &exception)
{
    QMessageBox::warning(0, QObject::tr("SAX Handler"),
                         QObject::tr("Parse error at line %1, column "
                                     "%2:\n%3.")
                         .arg(exception.lineNumber())
                         .arg(exception.columnNumber())
                         .arg(exception.message()));
    return false;
}
```

La fonction `fatalError()` est appelée lorsque le lecteur ne parvient pas à analyser le fichier XML. Dans cette situation, nous affichons simplement une boîte de message, en donnant le numéro de ligne, le numéro de colonne et le texte d'erreur de l'analyseur.

Ceci termine l'implémentation de la classe `SaxHandler`. Voyons maintenant comment l'utiliser :

```
bool parseFile(const QString &fileName)
{
    QStringList labels;
    labels << QObject::tr("Terms") << QObject::tr("Pages");

    QTreeWidget *treeWidget = new QTreeWidget;
    treeWidget->setHeaderLabels(labels);
    treeWidget->setWindowTitle(QObject::tr("SAX Handler"));
    treeWidget->show();
```

```
    QFile file(fileName);
    QXmlInputSource inputSource(&file);
    QXmlSimpleReader reader;
    SaxHandler handler(treeWidget);
    reader.setContentHandler(&handler);
    reader.setErrorHandler(&handler);
    return reader.parse(inputSource);
}
```

Nous définissons un `QTreeWidget` avec deux colonnes. Puis nous créons un objet `QFile` pour le fichier devant être lu et un `QXmlSimpleReader` pour analyser le fichier. Il n'est pas nécessaire d'ouvrir le `QFile` par nous-mêmes. `QXmlInputSource` s'en charge automatiquement.

Nous créons enfin un objet `SaxHandler`, nous l'installons sur le lecteur à la fois en tant que gestionnaire de contenu et en tant que gestionnaire d'erreur, et nous appelons `parse()` sur le lecteur pour effectuer l'analyse.

Au lieu de transmettre un simple objet de fichier à la fonction `parse()`, nous transmettons un `QXmlInputSource`. Cette classe ouvre le fichier qui lui est fourni, le lit (en prenant en considération tout codage de caractères spécifié dans la déclaration `<?xml?>`), et fournit une interface par le biais de laquelle l'analyseur lit le fichier.

Dans `SaxHandler`, nous réimplémentons uniquement les fonctions des classes `QXmlContentHandler` et `QXmlErrorHandler`. Si nous avions implémenté des fonctions d'autres classes gestionnaires, nous aurions également dû appeler leurs fonctions de réglage (set) sur le lecteur.

Pour lier l'application à la bibliothèque *QtXml*, nous devons ajouter cette ligne dans le fichier `.pro` :

```
QT           += xml
```

Lire du code XML avec DOM

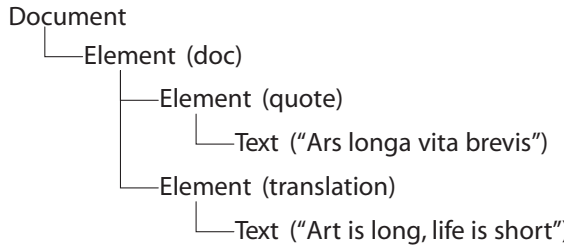
DOM est une API standard pour l'analyse de code XML développée par le W3C (*World Wide Web Consortium*). Qt fournit une implémentation DOM Niveau 2 destinée à la lecture, à la manipulation et à l'écriture de documents XML.

DOM présente un fichier XML sous la forme d'un arbre en mémoire. Nous pouvons parcourir l'arbre DOM autant que nécessaire. Il nous est également possible de le modifier et de le réenregistrer sur le disque en tant que fichier XML.

Considérons le document XML suivant :

```
<doc>
   <quote>Ars longa vita brevis</quote>
   <translation>Art is long, life is short</translation>
</doc>
```

Il correspond à l'arbre DOM ci-après :



L'arbre DOM contient des nœuds de types différents. Par exemple, un nœud `Element` correspond à une balise d'ouverture et à sa balise de fermeture. Le matériau situé entre les balises apparaît sous la forme de nœuds enfants de `Element`.

Dans Qt, les types de nœud (comme toutes les autres classes en liaison avec DOM) possèdent un préfixe `QDom`. Ainsi, `QDomElement` représente un nœud `Element`, et `QDomText` représente un nœud `Text`.

Chaque nœud peut posséder différents types de nœuds enfants. Par exemple, un nœud `Element` peut contenir d'autres nœuds `Element`, ainsi que des nœuds `EntityReference`, `Text`, `CDATASection`, `ProcessingInstruction` et `Comment`. La Figure 15.3 présente les types de nœuds enfants correspondant aux nœuds parents. Ceux apparaissant en gris ne peuvent pas avoir de nœud enfant.

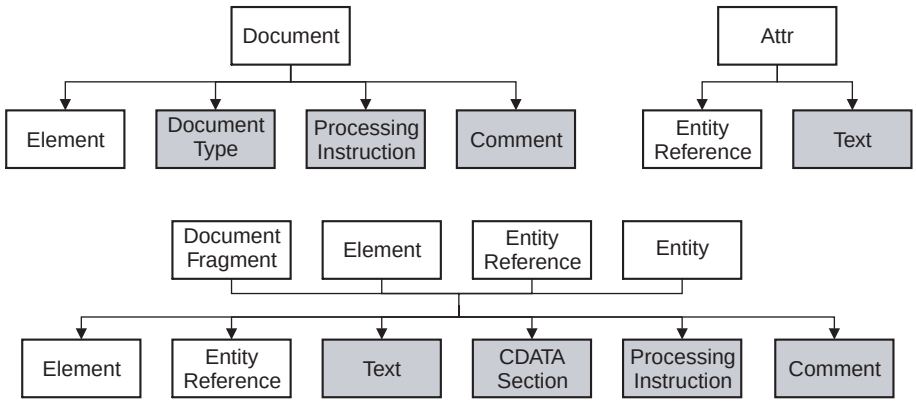


Figure 15.3
Relations parent/enfant entre les nœuds DOM

Nous allons voir comment utiliser DOM pour lire des fichiers XML en créant un analyseur pour le format de fichier d'index décrit dans la section précédente.

```
class DomParser
{
public:
   DomParser(QIODevice *device, QTreeWidget *tree);

private:
   void parseEntry(const QDomElement &element,
                   QTreeWidgetItem *parent);

   QTreeWidget *treeWidget;
};
```

Nous définissons une classe nommée `DomParser` qui analysera un index de livre se présentant sous la forme d'un document XML et affichera le résultat dans un `QTreeWidget`. Cette classe n'hérite d'aucune autre classe.

```
DomParser::DomParser(QIODevice *device, QTreeWidget *tree)
{
   treeWidget = tree;

   QString errorStr;
   int errorLine;
   int errorColumn;

   QDomDocument doc;
   if (!doc.setContent(device, true, &errorStr, &errorLine,
                       &errorColumn)) {
      QMessageBox::warning(0, QObject::tr("DOM Parser"),
                           QObject::tr("Parse error at line %1, "
                                       "column %2:\n%3")
                           .arg(errorLine)
                           .arg(errorColumn)
                           .arg(errorStr));
   return;
}

QDomElement root = doc.documentElement();
if (root.tagName() != "bookindex")
    return;

QDomNode node = root.firstChild();
while (!node.isNull()) {
   if (node.toElement().tagName() == "entry")
      parseEntry(node.toElement(), 0);
   node = node.nextSibling();
  }
}
```

Dans le constructeur, nous créons un objet `QDomDocument` et appelons `setContent()` sur celui-ci pour l'amener à lire le document XML fourni par le `QIODevice`. La fonction `setContent()` ouvre automatiquement le périphérique si ce n'est déjà fait. Nous appelons ensuite `documentElement()` sur le `QDomDocument` pour obtenir son enfant `QDomElement`

unique, et nous vérifions s'il s'agit bien d'un élément `<bookindex>`. Nous parcourons tous les nœuds enfants, et si le nœud est un élément `<entry>`, nous appelons `parseEntry()` pour l'analyser.

La classe `QDomNode` peut stocker tout type de nœud. Si nous souhaitons traiter un nœud de façon plus précise, nous devons tout d'abord le convertir en un type de donnée correct. Dans cet exemple, nous ne nous préoccupons que des nœuds `Element`. Nous appelons donc `toElement()` sur le `QDomNode` pour le convertir en un `QDomElement`, puis nous appelons `tagName()` afin de récupérer le nom de balise de l'élément. Si le nœud n'est *pas* du type `Element`, la fonction `toElement()` retourne un objet `QDomElement` nul, avec un nom de balise vide.

```
void DomParser::parseEntry(const QDomElement &element,
                           QTreeWidgetItem *parent)
{
   QTreeWidgetItem *item;
   if (parent) {
      item = new QTreeWidgetItem(parent);
   } else {
      item = new QTreeWidgetItem(treeWidget);
   }
   item->setText(0, element.attribute("term"));

   QDomNode node = element.firstChild();
   while (!node.isNull()) {
      if (node.toElement().tagName() == "entry") {
         parseEntry(node.toElement(), item);
      } else if (node.toElement().tagName() == "page") {
         QDomNode childNode = node.firstChild();
         while (!childNode.isNull()) {
            if (childNode.nodeType() == QDomNode::TextNode) {
               QString page = childNode.toText().data();
               QString allPages = item->text(1);
               if (!allPages.isEmpty())
                  allPages += ", ";
               allPages += page;
               item->setText(1, allPages);
               break;
            }
            childNode = childNode.nextSibling();
         }
      }
      node = node.nextSibling();
   }
}
```

Dans `parseEntry()`, nous créons un élément `QTreeWidget`. Si elle est imbriquée dans une autre balise `<entry>`, la nouvelle balise définit une sous-entrée dans l'index, et le nouveau `QTreeWidgetItem` est créé en tant qu'enfant du `QTreeWidgetItem` qui représente l'entrée

principale. Dans le cas contraire, nous créons le `QTreeWidgetItem` avec `treeWidget` en tant que parent, en faisant de celui-ci un élément de haut niveau. Nous appelons `setText()` pour définir le texte présenté en colonne 0 en la valeur de l'attribut `term` de la balise `<entry>`.

Une fois le `QTreeWidgetItem` initialisé, nous parcourons les nœuds enfants du `QDomElement` correspondant à la balise `<entry>` courante.

Si l'élément est `<entry>`, nous appelons `parseEntry()` avec l'élément courant en tant que deuxième argument. Le `QTreeWidgetItem` de la nouvelle entrée sera alors créé avec le `QTreeWidgetItem` de l'entrée principale en tant que parent.

Si l'élément est `<page>`, nous parcourons la liste enfant de l'élément à la recherche d'un nœud `Text`. Une fois celui-ci trouvé, nous appelons `Text()` pour le convertir en un objet `QDomText`, et `data()` pour extraire le texte en tant que `QString`. Puis nous ajoutons le texte à la liste de numéros de page délimitée par des virgules dans la colonne 1 du `QTreeWidgetItem`.

Voyons comment utiliser la classe `DomParser` pour analyser un fichier :

```
void parseFile(const QString &fileName)
{
   QStringList labels;
   labels << QObject::tr("Terms") << QObject::tr("Pages");

   QTreeWidget *treeWidget = new QTreeWidget;
   treeWidget->setHeaderLabels(labels);
   treeWidget->setWindowTitle(QObject::tr("DOM Parser"));
   treeWidget->show();

   QFile file(fileName);
   DomParser(&file, treeWidget);
}
```

Nous commençons par définir un `QTreeWidget`. Puis nous créons un `QFile` et un `DomParser`. Une fois le `DomParser` construit, il analyse le fichier et alimente l'arborescence.

Comme dans l'exemple précédent, nous avons besoin de la ligne suivante dans le fichier `.pro` de l'application pour établir un lien avec la bibliothèque *QtXml* :

```
QT         += xml
```

Comme le montre l'exemple, l'opération consistant à parcourir un arbre DOM peut s'avérer assez lourde. La simple extraction du texte entre les balises `<page>` et `</page>` a nécessité le parcours d'une liste de `QDomNode` au moyen de `firstChild()` et de `nextSibling()`. Les programmeurs qui utilisent beaucoup DOM écrivent souvent leurs propres fonctions conteneur de haut niveau afin de simplifier les opérations courantes, telles que l'extraction de texte entre les balises d'ouverture et de fermeture.

Ecrire du code XML

Deux approches s'offrent à vous pour générer des fichiers XML à partir d'applications Qt :

- générer un arbre DOM et appeler `save()` sur celui-ci ;
- générer le code XML manuellement.

Le choix entre ces approches est souvent indépendant du fait que nous utilisions SAX ou DOM pour la lecture des documents XML.

Voici un extrait de code qui illustre comment créer un arbre DOM et l'écrire au moyen d'un `QTextStream` :

```
const int Indent = 4;

QDomDocument doc;
QDomElement root = doc.createElement("doc");
QDomElement quote = doc.createElement("quote");
QDomElement translation = doc.createElement("translation");
QDomText latin = doc.createTextNode("Ars longa vita brevis");
QDomText english = doc.createTextNode("Art is long, life is short");
doc.appendChild(root);
root.appendChild(quote);
root.appendChild(translation);
quote.appendChild(latin);
translation.appendChild(english);

QTextStream out(&file);
doc.save(out, Indent);
```

Le deuxième argument de `save()` est la taille du retrait à utiliser. Une valeur différente de zéro facilite la lecture du contenu du fichier. Voici la sortie du fichier XML :

```
<doc>
   <quote>Ars longa vita brevis</quote>
   <translation>Art is long, life is short</translation>
</doc>
```

Un autre scénario se produit dans les applications qui utilisent l'arbre DOM comme structure de données primaire. Ces applications effectuent généralement des opérations de lecture dans des documents XML en utilisant DOM, puis modifient l'arbre DOM en mémoire et appellent finalement `save()` pour convertir de nouveau l'arbre vers XML.

Par défaut, `QDomDocument::save()` utilise l'encodage UTF-8 pour le fichier généré. Nous pouvons utiliser un autre encodage en ajoutant au début de l'arbre DOM une déclaration XML telle que :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

L'extrait de code suivant vous montre comment y parvenir :

```
QTextStream out(&file);
QDomNode xmlNode = doc.createProcessingInstruction("xml",
                           "version=\"1.0\" encoding=\"ISO-8859-1\"");
doc.insertBefore(xmlNode, doc.firstChild());
doc.save(out, Indent);
```

La génération manuelle de fichiers XML n'est pas beaucoup plus difficile qu'avec DOM. Nous pouvons employer `QTextStream` et écrire les chaînes comme nous le ferions avec tout autre fichier texte. La partie la plus délicate est de neutraliser l'interprétation des caractères spéciaux qui apparaissent dans le texte et les valeurs d'attribut. La fonction `Qt::escape()` neutralise les caractères `<`, `>` et `&`. Voici un extrait de code qui fait appel à cette fonction :

```
QTextStream out(&file);
out.setCodec("UTF-8");
out << "<doc>\n"
    << " <quote>" << Qt::escape(quoteText) << "</quote>\n"
    << " <translation>" << Qt::escape(translationText)
    << "</translation>\n"
    << "</doc>\n";
```

L'article "Generating XML" disponible à l'adresse **http://doc.trolltech.com/qq/qq05-generating-xml.html** présente une classe très simple qui facilite la génération de fichiers XML. Cette classe prend en charge les détails tels que les caractères spéciaux, le retrait et les problèmes d'encodage, nous permettant de nous concentrer librement sur le code XML à générer. La classe a été conçue pour fonctionner avec Qt 3, mais il n'est pas difficile de l'adapter à Qt 4.

16

Aide en ligne

Au sommaire de ce chapitre

- ✔ Infobulles, informations d'état et aide "Qu'est-ce que c'est ?"
- ✔ Utiliser `QTextBrowser` comme moteur d'aide simple
- ✔ Utiliser l'Assistant Qt pour une aide en ligne puissante

La plupart des applications fournissent une aide en ligne à leurs utilisateurs. Certaines indications apparaissent sous une forme brève, telles que les infobulles, les informations d'état et "Qu'est-ce que c'est ?". Qt prend naturellement en charge toutes ces informations. Il existe également un autre type d'aide, beaucoup plus approfondi, qui implique de nombreuses pages de texte. Dans ce cas, nous pouvons utiliser `QTextBrowser` en tant que navigateur d'aide simple. Il est également possible d'invoquer l'*Assistant Qt* ou un navigateur HTML depuis notre application.

Infobulles, informations d'état et aide "Qu'est-ce que c'est ?"

Une infobulle est un petit texte qui apparaît lorsque la souris survole un widget. Les infobulles sont présentées sous la forme de texte noir sur un arrière-plan jaune. Leur objectif principal est de fournir des descriptions textuelles de boutons des barres d'outils.

Nous pouvons ajouter des infobulles à des widgets arbitraires dans le code au moyen de `QWidget::setToolTip()`. Par exemple :

```
findButton->setToolTip(tr("Find next"));
```

Pour définir l'infobulle d'un `QAction` que vous ajoutez à un menu ou à une barre d'outils, nous appelons simplement `setToolTip()` sur l'action. Par exemple :

```
newAction = new QAction(tr("&New"), this);
newAction->setToolTip(tr("New document"));
```

Si nous ne définissons pas explicitement une infobulle, `QAction` utilise automatiquement le texte de l'action.

Une information d'état est également un texte descriptif bref, mais généralement un peu plus long que celui d'une infobulle. Lorsque la souris survole un bouton de barre d'outils ou une option de menu, une information d'état apparaît dans la barre d'état. (Voir Figure 16.1) Appelez `setStatusTip()` pour ajouter une information d'état à une action ou à un widget :

```
newAction->setStatusTip(tr("Create a new document"));
```

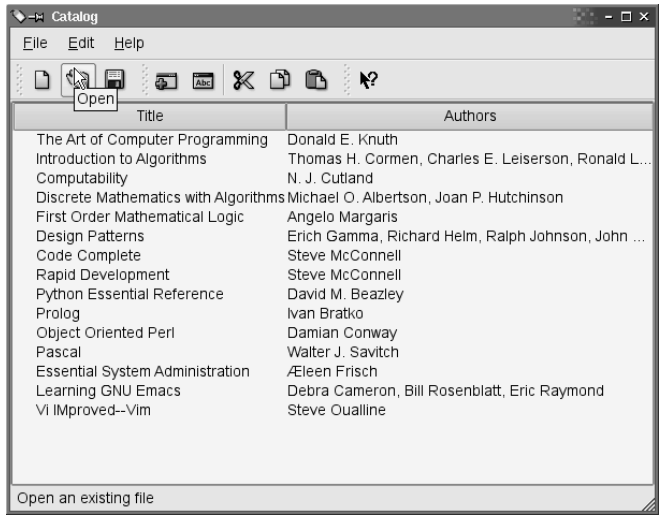


Figure 16.1
Une application affichant une infobulle et une information d'état

Dans certaines situations, il est souhaitable de fournir une information plus complète que celle offerte par les infobulles et les indicateurs d'état. Nous pouvons, par exemple, afficher une boîte de dialogue complexe contenant un texte explicatif concernant chaque champ, ceci sans forcer l'utilisateur à invoquer une fenêtre d'aide distincte. Le mode "Qu'est-ce que c'est ?" représente une solution idéale dans cette situation. Quand une fenêtre se trouve en mode "Qu'est-ce que c'est ?", le curseur se transforme en et l'utilisateur peut cliquer sur tout composant de l'interface utilisateur pour obtenir le texte d'aide le concernant. Pour entrer en mode "Qu'est-ce que c'est ?", l'utilisateur peut soit cliquer sur le bouton ? dans la barre de titre de la fenêtre (sous Windows et KDE), soit appuyer sur `Maj+F1`.

Voici un exemple de texte "Qu'est-ce que c'est ?" :

```
dialog->setWhatsThis(tr("<img src=\":/images/icon.png\">"
                        " The meaning of the Source field depends "
                        "on the Type field:"
                        "<ul>"
                        "<li><b>Books</b> have a Publisher"
                        "<li><b>Articles</b> have a Journal name with "
                        "volume and issue number"
                        "<li><b>Theses</b> have an Institution name "
                        "and a Department name"
                        "</ul>"));
```

Nous pouvons utiliser les balises HTML pour mettre en forme le texte d'information d'un "Qu'est-ce que c'est ?". Dans l'exemple fourni, nous incluons une image (qui est répertoriée dans le fichier de ressources de l'application), une liste à puces et du texte en gras. (Voir Figure 16.2.) Vous trouverez les balises et attributs pris en charge par Qt à l'adresse **http://doc.trolltech.com/4.1/richtext-html-subset.html**.

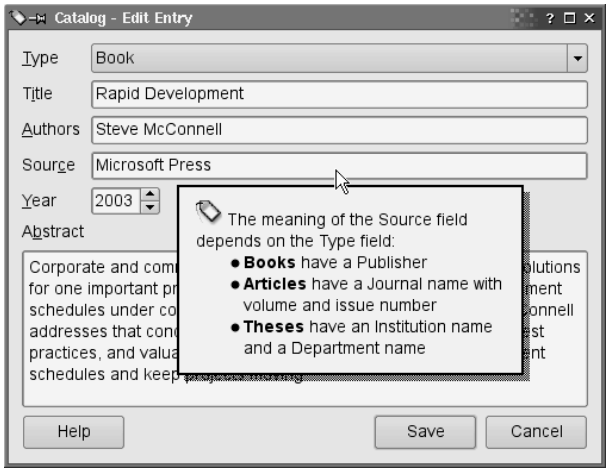


Figure 16.2
Une boîte de dialogue affichant du texte d'aide "Qu'est-ce que c'est ?"

Un texte "Qu'est-ce que c'est ?" défini sur une action apparaît quand l'utilisateur clique sur l'élément de menu ou sur le bouton de la barre d'outils, ou encore s'il appuie sur la touche de raccourci alors qu'il se trouve en mode "Qu'est-ce que c'est ?".Lorsque les composants de l'interface utilisateur de la fenêtre principale d'une application sont associés à du texte "Qu'est-ce que c'est ?", il est habituel de proposer une option de même nom dans le menu `Aide` et un bouton correspondant dans la barre d'outils. Pour ce faire, il suffit de créer une action `Qu'est-ce que c'est ?` avec la fonction statique `QWhatThis::createAction()` et d'ajouter l'action retournée à un menu `Aide` et à une barre d'outils. La classe `QWhatsThis` fournit des fonctions statiques destinées à programmer l'entrée dans le mode "Qu'est-ce que c'est ?" et la sortie de ce mode.

Utilisation de *QTextBrowser* comme moteur d'aide simple

Les grosses applications nécessitent souvent une aide en ligne plus riche que celle susceptible d'être offerte pas les infobulles, les informations d'état et le mode "Qu'est-ce que c'est ?". Une solution simple consiste à fournir un navigateur d'aide. Les applications incluant un navigateur de ce type possèdent généralement une entrée `Aide` dans le menu `Aide` de la fenêtre principale ainsi qu'un bouton `Aide` dans chaque boîte de dialogue.

Dans cette section, nous présentons le navigateur d'aide illustré en Figure 16.3 et expliquons comment l'utiliser dans une application. La fenêtre fait appel à `QTextBrowser` pour afficher les pages d'aide dont la syntaxe est basée sur HTML. `QTextBrowser` étant en mesure de gérer de nombreuses balises HTML, il s'avère idéal dans cette situation.

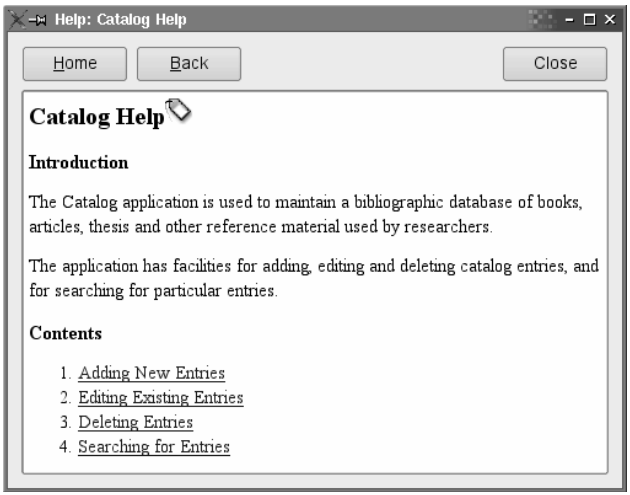
Nous commençons par le fichier d'en-tête :

```
#include <QWidget>

class QPushButton;
class QTextBrowser;

class HelpBrowser : public QWidget
{
   Q_OBJECT
public:
   HelpBrowser(const QString &path, const QString &page,
               QWidget *parent = 0);

   static void showPage(const QString &page);

private slots:
   void updateWindowTitle();
```

```
private:
   QTextBrowser *textBrowser;
   QPushButton *homeButton;
   QPushButton *backButton;
   QPushButton *closeButton;
};
```

`HelpBrowser` fournit une fonction statique pouvant être appelée n'importe où dans l'application. Cette fonction crée une fenêtre `HelpBrowser` et affiche la page donnée.

Figure 16.3
Le widget `HelpBrowser`



Voici le début de l'implémentation :

```
#include <QtGui>

#include "helpbrowser.h"

HelpBrowser::HelpBrowser(const QString &path, const QString &page,
                         QWidget *parent)
   : QWidget(parent)
{
   setAttribute(Qt::WA_DeleteOnClose);
   setAttribute(Qt::WA_GroupLeader);

   textBrowser = new QTextBrowser;

   homeButton = new QPushButton(tr("&Home"));
   backButton = new QPushButton(tr("&Back"));
   closeButton = new QPushButton(tr("Close"));
   closeButton->setShortcut(tr("Esc"));
```

```
    QHBoxLayout *buttonLayout = new QHBoxLayout;
    buttonLayout->addWidget(homeButton);
    buttonLayout->addWidget(backButton);
    buttonLayout->addStretch();
    buttonLayout->addWidget(closeButton);

    QVBoxLayout *mainLayout = new QVBoxLayout;
    mainLayout->addLayout(buttonLayout);
    mainLayout->addWidget(textBrowser);
    setLayout(mainLayout);

    connect(homeButton, SIGNAL(clicked()), textBrowser, SLOT(home()));
    connect(backButton, SIGNAL(clicked()),
            textBrowser, SLOT(backward()));
    connect(closeButton, SIGNAL(clicked()), this, SLOT(close()));
    connect(textBrowser, SIGNAL(sourceChanged(const QUrl &)),
            this, SLOT(updateWindowTitle()));

    textBrowser->setSearchPaths(QStringList() << path << ":/images");
    textBrowser->setSource(page);
}
```

Nous définissons l'attribut `Qt::WA_GroupLeader` car nous souhaitons faire apparaître les fenêtres `HelpBrowser` depuis des boîtes de dialogue modales en complément de la fenêtre principale. Ces boîtes empêchent normalement l'utilisateur d'interagir avec toute autre fenêtre de l'application. Cependant, après avoir demandé de l'aide, cet utilisateur doit de toute évidence être autorisé à interagir à la fois avec la boîte de dialogue modale et le navigateur d'aide. La définition de l'attribut `Qt::WA_GroupLeader` autorise cette interaction.

Nous fournissons deux accès pour la recherche, le premier étant le système de fichiers contenant la documentation de l'application et le second étant l'emplacement des ressources image. Le code HTML peut inclure des références aux images dans le système de fichiers de façon classique, mais il peut également faire référence aux ressources image en utilisant un chemin d'accès commençant par `:/` (deux points, slash). Le paramètre `page` est le nom du fichier de documentation, avec une ancre HTML facultative (une ancre HTML est la cible d'un lien).

```
void HelpBrowser::updateWindowTitle()
{
    setWindowTitle(tr("Help: %1").arg(textBrowser->documentTitle()));
}
```

Dès que la page source change, le slot `updateWindowTitle()` est appelé. La fonction `documentTitle()` retourne le texte spécifié dans la balise `<title>` de la page.

```
void HelpBrowser::showPage(const QString &page)
{
    QString path = QApplication::applicationDirPath() + "/doc";
    HelpBrowser *browser = new HelpBrowser(path, page);
```

```
    browser->resize(500, 400);
    browser->show();
}
```

Dans la fonction statique `showPage()`, nous créons la fenêtre `HelpBrowser`, puis nous l'affichons. Comme nous avons défini l'attribut `Qt::WA_DeleteOnClose` dans le constructeur de `HelpBrowser`, la fenêtre sera détruite automatiquement lors de sa fermeture par l'utilisateur.

Pour cet exemple, nous supposons que la documentation est stockée dans le sous-répertoire `doc` du répertoire contenant l'exécutable de l'application. Toutes les pages transmises à la fonction `showPage()` seront extraites de ce sous-répertoire.

Vous êtes maintenant prêt à invoquer le navigateur d'aide depuis l'application. Dans la fenêtre principale de l'application, créez une action `Aide` et connectez-la à un slot `help()` sur le modèle suivant :

```
void MainWindow::help()
{
    HelpBrowser::showPage("index.html");
}
```

Nous supposons ici que le fichier d'aide principal se nomme `index.html`. Dans le cas de boîtes de dialogue, vous connecteriez le bouton Aide à un slot `help()` similaire à celui-ci :

```
void EntryDialog::help()
{
    HelpBrowser::showPage("forms.html#editing");
}
```

Ici, nous effectuons la recherche dans un fichier d'aide différent, `forms.html` et faisons défiler le `QTextBrowser` jusqu'à l'ancre `editing`.

Utilisation de l'assistant pour une aide en ligne puissante

L'Assistant Qt est une application d'aide en ligne redistribuable fournie par Trolltech. Elle présente l'avantage de prendre en charge l'indexation et la recherche de texte intégral et d'être capable de gérer plusieurs jeux de documentation distincts correspondant à différentes applications.

Pour utiliser *l'Assistant Qt*, nous devons incorporer le code nécessaire dans notre application et faire connaître l'existence de notre documentation à cet assistant.

La communication entre une application Qt et *l'Assistant Qt* est gérée par la classe `QAssistantClient`, qui est située dans une bibliothèque distincte. Pour établir une liaison entre cette bibliothèque et une application, vous devez ajouter cette ligne de code au fichier `.pro` de l'application :

```
CONFIG     += assistant
```

Nous allons maintenant examiner le code d'une nouvelle classe `HelpBrowser` qui utilise *l'Assistant Qt*.

```
#ifndef HELPBROWSER_H
#define HELPBROWSER_H

class QAssistantClient;
class QString;

class HelpBrowser
{
public:
   static void showPage(const QString &page);

private:
   static QAssistantClient *assistant;
};

#endif
```

Voici le nouveau fichier `helpbrowser.cpp` :

```
#include <QApplication>
#include <QAssistantClient>

#include "helpbrowser.h"

QAssistantClient *HelpBrowser::assistant = 0;

void HelpBrowser::showPage(const QString &page)
{
   QString path = QApplication::applicationDirPath() + "/doc/" + page;
   if (!assistant)
       assistant = new QAssistantClient("");
   assistant->showPage(path);
}
```

Le constructeur de `QAssistantClient` accepte comme premier argument un chemin d'accès qu'il utilise pour situer l'exécutable de *Assistant Qt*. En transmettant un chemin d'accès vide, nous indiquons à `QAssistantClient` de rechercher l'exécutable dans la variable d'environnement `PATH`. `QAssistantClient` possède une fonction `showPage()` qui accepte un nom de page avec une ancre HTML en option.

La prochaine étape consiste à préparer une table des matières et un index pour la documentation. Pour ce faire, nous créons un profile *Assistant Qt* et écrivons un fichier `.dcf` qui fournit des informations concernant la documentation. Tout ceci est expliqué dans la documentation en ligne de *l'Assistant Qt*. Nous ne répéterons donc pas ces indications ici.

Une solution alternative à l'emploi de `QTextBrowser` ou à celui de *l'Assistant Qt* consiste à s'orienter vers des approches spécifiques à la plate-forme. Pour les applications Windows, il peut être souhaitable de créer des fichiers d'aide HTML Windows et d'offrir un accès à ceux-ci par le biais de Microsoft Internet Explorer. Pour ce faire, vous pouvez recourir à la classe

`QProcess` de Qt ou à l'infrastructure ActiveQt. L'approche la plus judicieuse pour les applications X11 serait de fournir des fichiers HTML et de lancer un navigateur Web au moyen de `QProcess`. Sous Mac OS X, l'Aide Apple fournit une fonctionnalité similaire à *l'Assistant Qt*.

Nous sommes à présent à la fin de la Partie II. Les chapitres de la Partie III traitent des fonctionnalités avancées et spécialisées de Qt. Le code C++ et Qt présenté n'est pas plus difficile que celui de la Partie II, mais certains concepts et idées vous paraîtront peut-être plus ardus, car ces domaines sont nouveaux pour vous.

III

Qt : étude avancée

17 *Internationalisation*
18 *Environnement multithread*
19 *Créer des plug-in*
20 *Fonctionnalités spécifiques à la plate-forme*
21 *Programmation embarquée*

17

Internationalisation

Au sommaire de ce chapitre

- ✔ Travailler avec Unicode
- ✔ Créer des applications ouvertes aux traductions
- ✔ Passer dynamiquement d'une langue à une autre
- ✔ Traduire les applications

En complément de l'alphabet latin utilisé pour le français et de nombreuses langues européennes, Qt offre une large prise en charge des systèmes d'écriture du reste du monde.

- Qt utilise Unicode en interne et par le biais de l'API. Ainsi, toutes les langues utilisées par l'interface utilisateur sont prises en charge de façon identique.
- Le moteur de texte de Qt est en mesure de gérer tous les systèmes d'écriture non-latins majeurs, dont l'arabe, le chinois, le cyrillique, l'hébreu, le japonais, le coréen, le thaï et les langues Hindi.
- Le moteur de disposition de Qt prend en charge l'écriture de la droite vers la gauche pour les langues telles que l'arabe et l'hébreu.
- Certaines langues nécessitent des méthodes spéciales d'entrée de texte. Les widgets d'édition tels que `QLineEdit` et `QTextEdit` fonctionnent correctement avec toute méthode d'entrée installée sur le système de l'utilisateur.

Il faut souvent fournir plus qu'une simple adaptation du texte saisi par les utilisateurs dans leur langue native. L'interface utilisateur entière doit également être traduite. Qt facilite cette tâche : il suffit de traiter toutes les chaînes visibles par l'utilisateur avec la fonction `tr()` (comme nous l'avons fait dans les chapitres précédents) et d'utiliser les outils de prise en charge de Qt pour préparer la traduction des fichiers dans les langues requises. Qt fournit un outil GUI nommé *Qt Linguist* destiné à être utilisé par les traducteurs. *Qt Linguist* est complété par deux programmes de ligne de commande, `lupdate` et `lrelease`, qui sont généralement exécutés par les développeurs de l'application.

Pour la plupart des applications, un fichier de traduction est chargé au démarrage qui tient compte des paramètres locaux de l'utilisateur. Mais dans certains cas, il est également nécessaire de pouvoir basculer d'une langue à l'autre au moment de l'exécution. Ceci est parfaitement possible avec Qt, bien que cette opération implique un travail supplémentaire. Et grâce au système de disposition de Qt, les divers composants de l'interface utilisateur sont automatiquement ajustés pour faire de la place aux textes traduits quand ils sont plus longs que les originaux.

Travailler avec Unicode

Unicode est un système de codage de caractères qui prend en charge la plupart des systèmes d'écriture mondiaux. L'idée à l'origine du développement d'Unicode est qu'en utilisant 16 bits au lieu de 8 pour stocker les caractères, il devient possible de coder environ 65 000 caractères au lieu de 256[1]. Unicode comprend les systèmes ASCII et ISO 8859-1 (Latin-1) et ces deux sous-ensembles se trouvent sur les mêmes positions de code. La valeur du caractère "A", par exemple, est de 0x41 dans les systèmes ASCII, Latin-1 et Unicode et celle de "Â" est de 0xD1 dans les systèmes Latin-1 et Unicode.

La classe `QString` de Qt stocke les chaînes utilisant le système Unicode. Chaque caractère d'un `QString` est un `QChar` de 16 bits et non un `char` de 8 bits. Voici deux méthodes pour définir le premier caractère d'une chaîne en "A" :

```
str[0] = 'A';
str[0] = QChar(0x41);
```

Si le fichier source est codé en Latin-1, il est aisé de spécifier des caractères en Latin-1 :

```
str[0] = 'Ñ';
```

Et si le codage du fichier source est différent, la valeur numérique fonctionne bien :

```
str[0] = QChar(0xD1);
```

1. Les versions récentes d'Unicode affectent des valeurs de caractères au-dessus de 65 535. Ces caractères peuvent être représentés avec des séquences de deux valeurs de 16 bits nommées "paires de substitution".

Nous pouvons désigner tout caractère Unicode par sa valeur numérique. Voici, par exemple, comment spécifier la lettre grecque majuscule sigma ("Σ") et le caractère monétaire euro ("€").

```
str[0] = QChar(0x3A3);
str[0] = QChar(0x20AC);
```

Les valeurs numériques de tous les caractères pris en charge par Unicode sont répertoriées à l'adresse **http://www.unicode.org/standard/**. Si votre besoin en caractères Unicode non-Latin-1 est rare et ponctuel, la recherche en ligne est la solution appropriée. Qt fournit cependant des moyens plus pratiques d'entrer des chaînes Unicode dans un programme, comme nous le verrons ultérieurement dans cette section.

Le moteur de texte de Qt 4 prend en charge les systèmes d'écriture suivants sur toutes les plates-formes : arabe, chinois, cyrillique, grec, hébreu, japonais, coréen, lao, latin, thaï et vietnamien. Il prend aussi en charge tous les scripts Unicode 4.1 ne nécessitant pas de traitement spécial. En outre, les systèmes d'écriture suivants sont pris en charge sur X11 avec Fonconfig et sur les versions récentes de Windows : bengali, devanagari, gujarati, gurmukhi, kannada, khmer, malayalam, syriac, tamil, telugu, thaana (dhivehi) et tibétain. L'oriya, enfin, est pris en charge sur X11 et le mongolien ainsi que le sinhala sont pris en charge par Windows XP. En supposant que les polices correctes sont installées sur le système, Qt peut afficher le texte au moyen de ces systèmes d'écriture. Et en supposant que les méthodes d'entrée correctes sont installées, les utilisateurs pourront entrer du texte correspondant à ces systèmes d'écriture dans leurs applications Qt.

La programmation avec `QChar` diffère légèrement de celle avec `char`. Pour obtenir la valeur numérique d'un `QChar`, vous devez appeler `unicode()` sur celui-ci. Pour obtenir la valeur ASCII ou Latin-1 d'un `QChar`, il vous faut appeler `toLatin1()`. Pour les caractères non-Latin-1, `toLatin1()` retourne "\0".

Si nous savons que toutes les chaînes d'un programme appartiennent au système ASCII, nous pouvons utiliser des fonctions `<cctype>` standard telles que `isalpha()`, `isdigit()` et `isspace()` sur la valeur de retour de `toLatin1()`. Il est cependant généralement préférable de faire appel aux fonctions membre de `QChar` pour réaliser ces opérations, car elles fonctionneront pour tout caractère Unicode. Les fonctions fournies par `QChar` incluent `isPrint()`, `isPunct()`, `isSpace()`, `isMark()`, `isLetter()`, `isNumber()`, `isLetterOrNumber()`, `isDigit()`, `isSymbol()`, `isLower()` et `isUpper()`. Voici, par exemple, un moyen de tester si un caractère est un chiffre ou une lettre majuscule :

```
if (ch.isDigit() || ch.isUpper())
   ...
```

L'extrait de code fonctionne pour tout alphabet qui distingue les majuscules des minuscules, dont l'alphabet latin, grec et cyrillique.

Lorsque vous avez une chaîne Unicode, vous pouvez l'utiliser à tout emplacement de l'API de Qt où est attendu un `QString`. Qt prend alors la responsabilité de l'afficher correctement et de la convertir en codages adéquats pour le système d'exploitation.

Nous devons être particulièrement attentifs lorsque nous lisons ou écrivons des fichiers texte. Ces derniers peuvent utiliser plusieurs codages, et il est souvent impossible de deviner le codage d'un fichier de ce type à partir de son contenu. Par défaut, `QTextStream` utilise le codage 8 bits local du système pour la lecture et l'écriture. Pour les Etats-Unis et l'Europe de l'ouest, il s'agit habituellement de Latin-1.

Si nous concevons notre propre format de fichier et souhaitons être en mesure de lire et d'écrire des caractères Unicode arbitraires, nous pouvons enregistrer les données sous la forme Unicode en appelant

```
stream.setCodec("UTF-16");
stream.setGenerateByteOrderMark(true);
```

avant de commencer l'écriture dans le `QTextStream`. Les données seront alors enregistrées au format UTF-16, format qui nécessite deux octets par caractère, et seront préfixées par une valeur spéciale de 16 bits (la marque d'ordre d'octet Unicode, `0xFFFE`) indiquant si ce fichier est en Unicode et si les octets se trouvent dans l'ordre little-endian ou big-endian. Le format UTF-16 étant identique à la représentation mémoire d'un `QString`, la lecture et l'écriture de chaînes Unicode dans ce format peut être très rapide. Il se produit cependant une surcharge lors de l'enregistrement de données ASCII pures au format UTF-16, car deux octets sont stockés pour chaque caractère au lieu d'un seul.

Il est possible de mentionner d'autres codage en appelant `setCodec()` avec un `QTextCodec` approprié. Un `QTextCodec` est un objet qui effectue une conversion entre Unicode et un codage donné. Les `QTextCodec` sont employés dans différents contextes par Qt. En interne, ils sont utilisés pour la prise en charge des polices, des méthodes d'entrée, du presse-papiers, des opérations de glisser-déposer et des noms de fichiers. Mais ils sont également utiles pour l'écriture d'applications Qt.

Lors de la lecture d'un fichier texte, `QTextStream` détecte Unicode automatiquement si le fichier débute par une marque d'ordre d'octet. Ce comportement peut être désactivé en appelant `setAutoDetectUnicode(false)`. Si les données ne sont pas supposées commencer par la marque d'ordre d'octet, il est préférable d'appeler `setCodec()` avec "UTF-16" avant la lecture.

UTF-8 est un autre codage qui prend en charge la totalité du système Unicode. Son principal avantage par rapport à UTF-16 est qu'il s'agit d'un super ensemble ASCII. Tout caractère se situant dans la plage `0x00` à `0x7F` est représenté par un seul octet. Les autres caractères, dont les caractères Latin-1 au-dessus de `0x7F`, sont représentés par des séquences multi-octets. Pour ce qui est du texte en majorité ASCII, UTF-8 occupe environ la moitié de l'espace consommé par UTF-16. Pour utiliser UTF-8 avec `QTextStream`, appelez `setCodec()` avec "UTF-8" comme nom de codec avant les opérations de lecture et d'écriture.

Si nous souhaitons toujours lire et écrire en Latin-1 sans tenir compte du système de codage local de l'utilisateur, nous pouvons définir le codec "ISO 8859-1" sur `QTextStream`. Par exemple :

```
QTextStream in(&file);
in.setCodec("ISO 8859-1");
```

Certains formats de fichiers spécifient leur codage dans leur en-tête. L'en-tête est généralement en ASCII brut pour assurer une lecture correcte quel que soit le codage utilisé. Le format de fichier XML en est un exemple intéressant. Les fichiers XML sont normalement encodés sous la forme UTF-8 ou UTF-16. Pour les lire correctement, il faut appeler `setCodec()` avec "UTF-8".Si le format est UTF-16, `QTextStream` le détectera automatiquement et s'adaptera. L'en-tête `<?xml?>` d'un fichier XML contient quelquefois un argument `encoding`. Par exemple :

```
<?xml version="1.0" encoding="EUC-KR"?>
```

Comme `QTextStream` ne vous permet pas de changer le codage une fois la lecture commencée, la meilleure façon d'appliquer un codage explicite consiste à recommencer à lire le fichier, en utilisant le codec correct (obtenu à l'aide de `QTextCodec::codecForName()`). Dans le cas de XML, nous pouvons éviter d'avoir à gérer le codage nous-mêmes en utilisant les classes XML de Qt décrites dans le Chapitre 15.

Les `QTextCodec` peuvent également être employés pour spécifier le codage de chaînes dans le code source. Considérons, par exemple, une équipe de programmeurs japonais écrivant une application destinée principalement au marché des particuliers. Il est probable que ces programmeurs écrivent leur code source dans un éditeur de texte qui utilise un codage tel que EUC-JP ou Shift-JIS. Un éditeur de ce type leur permet de saisir des caractères japonais sans problème. Ils peuvent donc écrire le type de code suivant :

```
QPushButton *button = new QPushButton(tr("日諾"));
```

Par défaut, Qt interprète les arguments de `tr()` comme Latin-1. Pour les autres cas, nous appelons la fonction statique `QTextCodec::setCodecForTr()`. Par exemple :

```
QTextCodec::setCodecForTr(QTextCodec::codecForName("EUC-JP"));
```

Cette opération doit être effectuée avant le premier appel à `tr()`. En général, elle est réalisée dans `main()`, immédiatement après la création de l'objet `QApplication`.

Les autres chaînes spécifiées dans le programme seront interprétées comme des chaînes Latin-1. Si les programmeurs souhaitent y entrer également des caractères japonais, ils peuvent les convertir explicitement en Unicode au moyen d'un `QTextCodec` :

```
QString text = japaneseCodec->toUnicode("海鮮料理");
```

Ils peuvent alternativement demander à Qt de faire appel à un codec spécifique lors de la conversion entre `const char*` et `QString` en appelant `QTextCodec::setCodecForCStrings()` :

```
QTextCodec::setCodecForCStrings(QTextCodec::codecForName("EUC-JP"));
```

Les techniques décrites ci-dessus peuvent être appliquées à toute langue non Latin-1, dont le chinois, le grec, le coréen et le russe.

Voici une liste des codages pris en charge par Qt 4 :

- Apple Roman
- Big5
- Big5-HKSCS
- EUC-JP
- EUC-KR
- GB18030-0
- IBM 850
- IBM 866
- IBM 874
- ISO 2022-JP
- ISO 8859-1
- ISO 8859-2
- ISO 8859-3
- ISO 8859-4
- ISO 8859-5
- ISO 8859-6
- ISO 8859-7
- ISO 8859-8
- ISO 8859-9
- ISO 8859-10
- ISO 8859-13
- ISO 8859-14
- ISO 8859-15
- ISO 8859-16
- Iscii-Bng
- Iscii-Dev
- Iscii-Gjr
- Iscii-Knd
- Iscii-Mlm
- Iscii-Ori
- Iscii-Pnj
- Iscii-Tlg
- Iscii-Tml
- JIS X 0201
- JIS X 0208
- KOI8-R
- KOI8-U
- MuleLao-1
- ROMAN8
- Shift-JIS
- TIS-620
- TSCII
- UTF-8
- UTF-16
- UTF-16BE
- UTF-16LE
- Windows-1250
- Windows-1251
- Windows-1252
- Windows-1253
- Windows-1254
- Windows-1255
- Windows-1256
- Windows-1257
- Windows-1258
- WINSAMI2

Pour tous ces codages, `QTextCodex::codecForName()` retournera toujours un pointeur valide. Les autres codages peuvent être pris en charge en dérivant `QTextCodec`.

Créer des applications ouvertes aux traductions

Pour que nos applications soient disponibles dans plusieurs langues, il convient de veiller à deux points :

- S'assurer que chaque chaîne visible par l'utilisateur passe par `tr()`.
- Charger un fichier de traduction (`.qm`) au démarrage.

Aucune de ces opérations n'est nécessaire pour les applications qui ne seront jamais traduites. Mais l'emploi de `tr()` ne nécessite pratiquement aucun effort et laisse la porte ouverte à toute traduction ultérieure.

`tr()` est une fonction statique définie dans `QObject` et remplacée dans chaque sous-classe définie avec la macro `Q_OBJECT`. Lorsque nous écrivons du code dans une sous-classe `QObject`, nous pouvons appeler `tr()` sans formalité. Un appel à `tr()` retourne une traduction si elle est disponible. Dans le cas contraire, le texte original est retourné.

Pour préparer les fichiers de traduction, nous devons exécuter l'outil `lupdate` de Qt. Cet outil extrait tous les littéraux de chaîne qui apparaissent dans les appels de `tr()` et produit des fichiers de traduction qui contiennent toutes les chaînes prêtes à être traduites. Les fichiers peuvent alors être expédiés à un traducteur afin qu'il y ajoute les traductions. Ce processus est expliqué dans la section "Traduction d'application" un peu plus loin dans ce chapitre.

La syntaxe d'un appel de `tr()` est la suivante :

```
Context::tr(sourceText, comment)
```

La partie `Context` est le nom d'une sous-classe `QObject` définie avec la macro `Q_OBJECT`. Il n'est pas nécessaire de lui préciser si nous appelons `tr()` depuis une fonction membre de la classe en question. La partie `sourceText` est le littéral chaîne à traduire. La partie `comment` est facultative. Elle permet de fournir des informations supplémentaires au traducteur.

Voici quelques exemples :

```
RockyWidget::RockyWidget(QWidget *parent)
   : QWidget(parent)
{
   QString str1 = tr("Letter");
   QString str2 = RockyWidget::tr("Letter");
   QString str3 = SnazzyDialog::tr("Letter");
   QString str4 = SnazzyDialog::tr("Letter", "US paper size");
}
```

Le contexte des deux premiers appels à `tr()` est "RockyWidget" et celui des deux derniers appels est "SnazzyDialog"."Letter" est le texte source des quatre appels. Le dernier d'entre eux comporte également un commentaire destiné à aider le traducteur à comprendre le sens du texte source.

Dans des contextes différents, les chaînes sont traduites indépendamment les unes des autres. Les traducteurs travaillent généralement sur un seul contexte à la fois, souvent avec l'application en cours d'exécution et en affichant la boîte de dialogue ou le widget soumis à la traduction.

Lorsque nous appelons `tr()` depuis une fonction globale, nous devons spécifier le contexte explicitement. Toute sous-classe `QObject` de l'application peut être employée en tant que contexte. Si aucun contexte n'est approprié, il est toujours possible de recourir à `QObject` lui-même. Par exemple :

```
int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   ...
   QPushButton button(QObject::tr("Hello Qt!"));
   button.show();
   return app.exec();
}
```

Dans tous les exemples étudiés jusqu'à présent, le contexte était celui d'un nom de classe. C'est pratique, car nous pouvons presque toujours l'omettre, mais ce n'est pas une obligation.

La façon la plus courante de traduire une chaîne en Qt consiste à utiliser la fonction `QApplication::translate()`, qui accepte jusqu'à trois arguments : le contexte, le texte source et le commentaire facultatif. Voici, par exemple, une autre façon de traduire "Hello Qt !":

```
QApplication::translate("Global Stuff", "Hello Qt!")
```

Ici, nous plaçons le texte dans le contexte "Global Stuff".

L'usage des fonctions `tr()` et `translate()` est double : elles remplissent à la fois le rôle des marqueurs utilisés par `lupdate` pour trouver les chaînes visibles par l'utilisateur, et elles agissent en tant que fonctions C++ qui traduisent du texte. Cette caractéristique a un impact sur la façon dont nous écrivons le code. Les lignes suivantes, par exemple, ne fonctionneront pas :

```
// INCORRECT
const char *appName = "OpenDrawer 2D";
QString translated = tr(appName);
```

Le problème ici est que `lupdate` ne sera pas en mesure d'extraire le littéral chaîne "OpenDrawer 2D", car il n'apparaît pas à l'intérieur d'un appel `tr()`. Le traducteur n'aura donc pas la possibilité de traduire la chaîne. Ce problème se pose souvent avec les chaînes dynamiques :

```
// INCORRECT
statusBar()->showMessage(tr("Host " + hostName + " found"));
```

Ici, la chaîne que nous transmettons à `tr()` varie en fonction de la valeur de `hostName`, de sorte que nous ne pouvons pas raisonnablement nous attendre à ce que `tr()` la traduise correctement.

La solution consiste à exécuter `QString::arg()` :

```
statusBar()->showMessage(tr("Host %1 found").arg(hostName));
```

Ce code repose sur le principe suivant : le littéral chaîne "Host %1 found" est transmis à `tr()`. En supposant qu'un fichier de traduction en français est chargé, `tr()` retournera quelque chose comme "Hôte %1 trouvé".Puis le paramètre "%1" est remplacé par le contenu de la variable `hostName`.

Bien qu'il soit généralement déconseillé d'appeler `tr()` sur une variable, il est possible de faire fonctionner cette technique correctement. Nous devons utiliser la macro `QT_TR_NOOP()` afin de marquer les littéraux chaîne à traduire avant de les affecter à une variable. Cette méthode se révèle particulièrement intéressante pour les tableaux statiques de chaînes. Par exemple :

```
void OrderForm::init()
{
   static const char * const flowers[] = {
      QT_TR_NOOP("Medium Stem Pink Roses"),
      QT_TR_NOOP("One Dozen Boxed Roses"),
      QT_TR_NOOP("Calypso Orchid"),
      QT_TR_NOOP("Dried Red Rose Bouquet"),
      QT_TR_NOOP("Mixed Peonies Bouquet"),
      0
```

```
    };

    for (int i = 0; flowers[i]; ++i)
       comboBox->addItem(tr(flowers[i]));
}
```

La macro `QT_TR_NOOP()` retourne simplement son argument. Mais `lupdate` extraira toutes les chaînes encadrées par cette dernière afin qu'elles puissent être traduites. Par la suite, au moment d'utiliser la variable, nous appellerons normalement `tr()`. Même si elle reçoit une variable, `cette fonction` remplit correctement son rôle de traduction.

La macro `QT_TRANSLATE_NOOP()` fonctionne comme `QT_TR_NOOP()` à la différence qu'elle reçoit aussi un contexte. Cette macro est pratique pour initialiser des variables à l'extérieur d'une classe :

```
static const char * const flowers[] = {
   QT_TRANSLATE_NOOP("OrderForm", "Medium Stem Pink Roses"),
   QT_TRANSLATE_NOOP("OrderForm", "One Dozen Boxed Roses"),
   QT_TRANSLATE_NOOP("OrderForm", "Calypso Orchid"),
   QT_TRANSLATE_NOOP("OrderForm", "Dried Red Rose Bouquet"),
   QT_TRANSLATE_NOOP("OrderForm", "Mixed Peonies Bouquet"),
   0
};
```

L'argument de contexte doit être identique au contexte fourni ultérieurement à `tr()` ou à `translate()`.

Lorsque nous commençons à utiliser `tr()` dans une application, le risque est grand d'oublier d'insérer des chaînes visibles par l'utilisateur dans un appel de cette fonction. Si ces appels manquants ne sont pas détectés par le traducteur, les utilisateurs de l'application vont voir apparaître certaines chaînes dans la langue originale. Pour éviter ce problème, nous pouvons demander à Qt d'interdire les conversions implicites de `const char*` en `QString`. Pour ce faire, nous définissons le symbole de préprocesseur `QT_NO_CAST_FROM_ASCII` avant d'inclure tout en-tête Qt. Le moyen le plus facile de s'assurer que ce symbole est défini consiste à ajouter la ligne suivante au fichier `.pro` de l'application :

```
DEFINES     += QT_NO_CAST_FROM_ASCII
```

Chaque littéral chaîne devra ainsi être obligatoirement traité par `tr()` ou `QLatin1String()`, selon qu'il devra être traduit ou non. Les chaînes qui ne seront pas encadrées par ces fonctions vont ainsi générer une erreur à la compilation, et il ne vous restera plus qu'à ajouter les appels de `tr()` ou `QLatin1String()` manquants.

Une fois chaque chaîne visible par l'utilisateur insérée dans un appel de `tr()`, il ne reste plus qu'à charger un fichier de traduction. L'opération se déroule généralement dans la fonction `main()` de l'application. Voici, par exemple, comment nous chargerions un fichier de traduction basé sur les paramètres locaux de l'utilisateur :

```
int main(int argc, char *argv[])
{
```

```
    QApplication app(argc, argv);
    QTranslator appTranslator;
    appTranslator.load("myapp_" + QLocale::system().name(),
                       qApp->applicationDirPath());
    app.installTranslator(&appTranslator);
    ...
    return app.exec();
}
```

La fonction `QLocale::System()` retourne un objet `QLocale` qui fournit des informations concernant les paramètres locaux de l'utilisateur. Par convention, nous intégrons le nom des paramètres locaux au nom du fichier `.qm`. Ces noms peuvent être plus ou moins précis : `fr`, par exemple, indique des paramètres locaux en langue française, `fr_CA` représente des paramètres locaux en français canadien et `fr_CA.ISO8859-15` en français canadien avec du codage ISO 8859-15 (un codage qui prend en charge "_", "Œ" et "œ").

En supposant que les paramètres locaux soient en `fr_CA.ISO8859-15`, la fonction `QTranslator::load()` essaie tout d'abord de charger le fichier `myapp_fr_CA.ISO8859-15.qm`. Si le fichier n'existe pas, `load()` essaie ensuite `myapp_fr_CA.qm`, puis `myapp_fr.qm` et enfin `myapp.qm` avant d'abandonner. Nous ne fournissons normalement qu'un fichier `myapp_fr.qm`, contenant une traduction en français standard, mais si nous souhaitons un fichier différent pour le français canadien, nous pouvons aussi fournir un fichier `myapp_fr_CA.qm` qui sera utilisé pour les paramètres locaux `fr_CA`.

Le deuxième argument de `QTranslator::load()` est le répertoire où nous souhaitons que `load()` recherche le fichier de traduction. Dans ce cas, nous supposons que les fichiers de traduction sont situés dans le même répertoire que l'exécutable.

Les bibliothèques Qt contiennent quelques chaînes nécessitant une traduction. Trolltech fournit des traductions en français, en allemand et en chinois simplifié dans le répertoire `translations` de Qt. Quelques autres langues sont également fournies, mais par les utilisateurs Qt. Ils ne sont pas officiellement pris en charge. Le fichier de traduction des bibliothèques Qt doit également être chargé.

```
QTranslator qtTranslator;
qtTranslator.load("qt_" + QLocale::system().name(),
                  qApp->applicationDirPath());
app.installTranslator(&qtTranslator);
```

Un objet `QTranslator` ne peut contenir qu'un seul fichier de traduction à la fois. C'est pourquoi nous utilisons un `QTranslator` distinct pour la traduction de Qt. Cela ne présente aucun problème puisque nous pouvons installer autant de traducteurs que nécessaire. `QApplication` les utilisera tous lors de la recherche d'une traduction.

Certaines langues, telles que l'arabe et l'hébreu, sont écrites de droite à gauche au lieu de gauche à droite. Dans cette situation, la mise en forme complète de l'application doit être inversée en appelant `QApplication::setLayoutDirection( Qt::RightToLeft)`. Les fichiers de traduction de Qt contiennent un marqueur spécial nommé "LTR" qui indique si la

langue s'écrit de gauche à droite ou de droite à gauche. Nous n'avons donc généralement pas besoin d'appeler `setLayoutDirection()`.

Il serait plus pratique pour les utilisateurs de fournir les applications avec les fichiers de traduction intégrés à l'exécutable, en utilisant le système de ressource de Qt. Cette technique permettrait non seulement de réduire le nombre de fichiers distribués pour constituer le produit, mais aussi d'éviter le risque de perte ou de suppression accidentelle de fichiers de traduction.

En supposant que les fichiers `.qm` sont situés dans un sous-répertoire `translations` se trouvant dans l'arbre source, nous aurions alors un fichier `myapp.qrc` avec le contenu suivant :

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
   <file>translations/myapp_de.qm</file>
   <file>translations/myapp_fr.qm</file>
   <file>translations/myapp_zh.qm</file>
   <file>translations/qt_de.qm</file>
   <file>translations/qt_fr.qm</file>
   <file>translations/qt_zh.qm</file>
</qresource>
</RCC>
```

Le fichier `.pro` contiendrait l'entrée suivante :

```
RESOURCES    = myapp.qrc
```

Nous devons enfin spécifier `:/translations` comme chemin d'accès aux fichiers de traduction dans `main()`. Les deux points qui apparaissent en première position indiquent que le chemin d'accès fait référence à une ressource et non à un fichier situé dans le système de fichiers.

Nous avons maintenant étudié tous les points nécessaires pour permettre à une application de fonctionner en utilisant des traductions dans d'autres langues. Mais la langue et la direction du système d'écriture ne sont pas les seuls points variables entre différents pays et cultures. Un programme internationalisé doit également prendre en compte les formats de date, d'heure, monétaire, numérique et l'ordre de classement des chaînes. Qt inclut une classe `QLocale` qui fournit des formats de date/d'heure et numérique localisés. Pour obtenir d'autres informations locales, nous pouvons faire appel aux fonctions C++ `setlocale()` et `localeconv()`.

Certaines classes et fonctions de Qt adaptent leur comportement en fonction des paramètres locaux :

- `QString::localeAwareCompare()` compare deux chaînes en prenant en compte les paramètres locaux. Elle permet de trier les éléments visibles par l'utilisateur.
- La fonction `toString()` fournie par `QDate`, `QTime` et `QDateTime` retourne une chaîne dans un format local quand elle est appelée avec `Qt::LocalDate` comme argument.
- Par défaut, les widgets `QDateEdit` et `QDateTimeEdit` présentent les dates dans le format local.

Enfin, il est possible qu'une application traduite ait besoin d'utiliser des icônes différentes de celles fournies initialement. Par exemple, les flèches gauche et droite apparaissant sur les

boutons `Précédente` et `Suivante` d'un navigateur Web doivent être inversées dans le cas d'une langue s'écrivant de droite à gauche. Voici comment procéder :

```
if (QApplication::isRightToLeft()) {
   backAction->setIcon(forwardIcon);
   forwardAction->setIcon(backIcon);
} else {
   backAction->setIcon(backIcon);
   forwardAction->setIcon(forwardIcon);
}
```

Les icônes contenant des caractères alphabétiques doivent très souvent être traduites. Par exemple, la lettre "I" qui apparaît sur un bouton de barre d'outils associé à une option `Italique` d'un traitement de texte doit être remplacée par un "C" en espagnol (`Cursivo`) et par un "K" en danois, néerlandais, allemand, norvégien et suédois (`Kursiv`). Voici un moyen simple d'y parvenir :

```
if (tr("Italic")[0] == 'C') {
   italicAction->setIcon(iconC);
} else if (tr("Italic")[0] == 'K') {
   italicAction->setIcon(iconK);
} else {
   italicAction->setIcon(iconI);
}
```

Une alternative consiste à utiliser la prise en charge de multiples paramètres locaux de la part du système de ressource. Dans le fichier `.qrc`, il est possible de spécifier un paramètre régional pour une ressource au moyen de l'attribut `lang`. Par exemple :

```
<qresource>
   <file>italic.png</file>
</qresource>
<qresource lang="es">
   <file alias="italic.png">cursivo.png</file>
</qresource>
<qresource lang="sv">
   <file alias="italic.png">kursiv.png</file>
</qresource>
```

Si le paramètre local de l'utilisateur est `es` (Español), `:/italic.png` fait alors référence à l'image `cursivo.png`. Si le paramètre local est `sv` (Svenska), c'est l'image `kursiv.png` qui est employée. Pour d'autres paramètres locaux, `italic.png` est utilisé.

Passer dynamiquement d'une langue à une autre

Pour la plupart des applications, la détection de la langue préférée de l'utilisateur dans `main()` et le chargement des fichiers `.qm` appropriés donne un résultat satisfaisant. Mais il existe des situations dans lesquelles les utilisateurs doivent pouvoir basculer dynamiquement d'une

langue à l'autre. Un changement de langue sans redémarrage peut être nécessaire pour une application employée en permanence par différentes personnes. Par exemple, les applications employées par les opérateurs de centres d'appels, par des traducteurs simultanés et par des opérateurs de caisses enregistreuses informatisées requièrent souvent cette possibilité.

Le passage dynamique d'une langue à une autre est un peu plus complexe à programmer que le chargement d'une traduction unique au lancement de l'application, mais ce n'est pas très difficile. Voici comment procéder :

- Vous devez offrir à l'utilisateur le moyen de passer à une autre langue.
- Pour chaque widget ou boîte de dialogue, vous devez définir toutes les chaînes susceptibles d'être traduites dans une fonction distincte (souvent nommée `retranslateUi()`) et appeler cette fonction lors des changements de langue.

Examinons les segments les plus importants du code source d'une application "centre d'appel".L'application fournit un menu `Language` permettant à l'utilisateur de définir la langue au moment de l'exécution. La langue par défaut est l'anglais. (Voir Figure 17.1)

Figure 17.1
Un menu `Language` *dynamique*

Comme nous ne savons pas quelle langue l'utilisateur souhaite employer au lancement de l'application, nous ne chargeons pas de traduction dans la fonction `main()`. Nous les chargeons plutôt dynamiquement quand elles s'avèrent nécessaires. Ainsi, tout le code dont nous avons besoin pour gérer les traductions doit être placé dans les classes des boîtes de dialogue et de la fenêtre principale.

Examinons la sous-classe `QMainWindow` de l'application.

```
MainWindow::MainWindow()
{
   journalView = new JournalView;
   setCentralWidget(journalView);

   qApp->installTranslator(&appTranslator);
   qApp->installTranslator(&qtTranslator);
   qmPath = qApp->applicationDirPath() + "/translations";

   createActions();
   createMenus();

   retranslateUi();
}
```

Dans le constructeur, nous définissons le widget central en tant que `JournalView`, une sous-classe de `QTableWidget`. Puis nous définissons quelques variables membre privées en liaison avec la traduction :

- La variable `appTranslator` est un objet `QTranslator` utilisé pour stocker la traduction de l'application en cours.
- La variable `qtTranslator` est un objet `QTranslator` utilisé pour stocker la traduction de Qt.
- La variable `qmPath` est un `QString` qui spécifie le chemin d'accès au répertoire contenant les fichiers de traduction de l'application.

Nous appelons enfin les fonctions privées `createActions()` et `createMenus()` pour créer le système de menus, et nous appelons `retranslateUi()`, également une fonction privée, pour définir les chaînes initialement visibles par l'utilisateur :

```
void MainWindow::createActions()
{
   newAction = new QAction(this);
   connect(newAction, SIGNAL(triggered()), this, SLOT(newFile()));
   ...
   aboutQtAction = new QAction(this);
   connect(aboutQtAction, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
}
```

La fonction `createActions()` crée normalement ses objets `QAction`, mais sans définir aucun texte ou touche de raccourci. Ces opérations seront effectuées dans `retranslateUi()`.

```
void MainWindow::createMenus()
{
   fileMenu = new QMenu(this);
   fileMenu->addAction(newAction);
   fileMenu->addAction(openAction);
   fileMenu->addAction(saveAction);
   fileMenu->addAction(exitAction);
   ...
   createLanguageMenu();

   helpMenu = new QMenu(this);
   helpMenu->addAction(aboutAction);
   helpMenu->addAction(aboutQtAction);

   menuBar()->addMenu(fileMenu);
   menuBar()->addMenu(editMenu);
   menuBar()->addMenu(reportsMenu);
   menuBar()->addMenu(languageMenu);
   menuBar()->addMenu(helpMenu);
}
```

La fonction `createMenus()` crée des menus, mais ne leur affecte aucun titre. Une fois encore, cette opération sera effectuée dans `retranslateUi()`.

Au milieu de la fonction, nous appelons `createLanguageMenu()` pour remplir le menu `Language` avec la liste des langues prises en charge. Nous reviendrons sur son code source dans un moment. Examinons tout d'abord `retranslateUi()` :

```
void MainWindow::retranslateUi()
{
   newAction->setText(tr("&New"));
   newAction->setShortcut(tr("Ctrl+N"));
   newAction->setStatusTip(tr("Create a new journal"));
   ...
   aboutQtAction->setText(tr("About &Qt"));
   aboutQtAction->setStatusTip(tr("Show the Qt library's About box"));

   fileMenu->setTitle(tr("&File"));
   editMenu->setTitle(tr("&Edit"));
   reportsMenu->setTitle(tr("&Reports"));
   languageMenu->setTitle(tr("&Language"));
   helpMenu->setTitle(tr("&Help"));
   setWindowTitle(tr("Call Center"));
}
```

C'est dans la fonction `retranslateUi()` que se produisent tous les appels de `tr()` pour la classe `MainWindow`. Ces appels ont lieu à la fin du constructeur de `MainWindow` et à chaque fois qu'un utilisateur change la langue de l'application par le biais du menu `Language`.

Nous définissons chaque texte, touche de raccourci et information d'état de `QAction`. Nous définissons également chaque titre de `QMenu` ainsi que le titre de fenêtre.

La fonction `createMenus()` présentée précédemment a appelé `createLanguageMenu()` pour remplir le menu `Language` avec une liste de langues :

```
void MainWindow::createLanguageMenu()
{
   languageMenu = new QMenu(this);

   languageActionGroup = new QActionGroup(this);
   connect(languageActionGroup, SIGNAL(triggered(QAction *)),
           this, SLOT(switchLanguage(QAction *)));

   QDir dir(qmPath);
   QStringList fileNames =
           dir.entryList(QStringList("callcenter_*.qm"));

   for (int i = 0; i < fileNames.size(); ++i) {
      QString locale = fileNames[i];
      locale.remove(0, locale.indexOf('_') + 1);
      locale.truncate(locale.lastIndexOf('.'));

      QTranslator translator;
      translator.load(fileNames[i], qmPath);
      QString language = translator.translate("MainWindow",
                                              "English");
```

```
    QAction *action = new QAction(tr("&%1 %2")
                                  .arg(i + 1).arg(language), this);
    action->setCheckable(true);
    action->setData(locale);

    languageMenu->addAction(action);
    languageActionGroup->addAction(action);

    if (language == "English")
       action->setChecked(true);
    }
}
```

Au lieu de coder en dur les langues prises en charge par l'application, nous créons une entrée de menu pour chaque fichier `.qm` situé dans le répertoire `translations` de l'application. Pour des raisons de simplicité, nous supposons que la langue anglaise (English) possède aussi un fichier `.qm`. Une alternative consisterait à appeler `clear()` sur les objets `QTranslator` lorsque l'utilisateur choisit English.

La difficulté consiste à trouver un nom adéquat pour la langue fournie par chaque fichier `.qm`. Le fait d'afficher simplement "en" pour "English" ou "de" pour "Deutsch" semble primaire et risque de semer la confusion dans l'esprit de certains utilisateurs. La solution retenue dans `createLanguageMenu()` est de contrôler la traduction de la chaîne "English" dans le contexte "MainWindow".Cette chaîne doit être transformée en "Deutsch" pour une traduction allemande et en "日本語" pour une traduction japonaise.

Nous créons un `QAction` à cocher pour chaque langue et stockons le nom local dans l'élément "data" de l'action. Nous les ajoutons à un objet `QActionGroup` afin de nous assurer qu'un seul élément du menu `Language` est coché à la fois. Quand une action du groupe est choisie par l'utilisateur, le `QActionGroup` émet le signal `triggered(QAction *)`, qui est connecté à `switchLanguage()`.

```
void MainWindow::switchLanguage(QAction *action)
{
   QString locale = action->data().toString();
   appTranslator.load("callcenter_" + locale, qmPath);
   qtTranslator.load("qt_" + locale, qmPath);
   retranslateUi();
}
```

Le slot `switchLanguage()` est appelé lorsque l'utilisateur choisit une langue dans le menu `Language`. Nous chargeons les fichiers de traduction de l'application et de Qt, et nous appelons `retranslateUi()` pour traduire toutes les chaînes de la fenêtre principale.

Sur Windows, une solution alternative au menu `Language` consisterait à répondre à des événements `LocaleChange`, un type d'événement émis par Qt quand un changement dans les paramètres locaux de l'environnement est détecté. Ce type existe sur toutes les plates-formes prises en charge par Qt, mais il n'est en fait généré que sur Windows, lorsque l'utilisateur change les paramètres locaux du système (dans les Options régionales et linguistiques du Panneau de

configuration). Pour gérer les événements `LocaleChange`, nous pouvons réimplémenter `QWidget::changeEvent()` comme suit :

```
void MainWindow::changeEvent(QEvent *event)
{
   if (event->type() == QEvent::LocaleChange) {
      appTranslator.load("callcenter_"
                         + QLocale::system().name(), qmPath);
      qtTranslator.load("qt_" + QLocale::system().name(), qmPath);
      retranslateUi();
   }
   QMainWindow::changeEvent(event);
}
```

Si l'utilisateur change les paramètres locaux pendant que l'application est en cours d'exécution, nous tentons de charger les fichiers de traduction corrects pour les nouveaux paramètres et appelons `retranslateUi()` pour mettre à jour l'interface utilisateur. Dans tous les cas, nous transmettons l'événement à la fonction `changeEvent()` de la classe de base, cette dernière pouvant aussi être intéressée par `LocaleChange` ou d'autres événements de changement.

Nous en avons maintenant terminé avec l'examen du code de `MainWindow`. Nous allons à présent observer le code de l'une des classes de widget de l'application, la classe `JournalView`, afin de déterminer quelles modifications sont nécessaires pour lui permettre de prendre en charge la traduction dynamique.

```
JournalView::JournalView(QWidget *parent)
   : QTableWidget(parent)
{
   ...
   retranslateUi();
}
```

La classe `JournalView` est une sous-classe de `QTableWidget`. A la fin du constructeur, nous appelons la fonction privée `retranslateUi()` pour définir les chaînes des widgets. Cette opération est similaire à celle que nous avons effectuée pour `MainWindows`.

```
void JournalView::changeEvent(QEvent *event)
{
   if (event->type() == QEvent::LanguageChange)
      retranslateUi();
   QTableWidget::changeEvent(event);
}
```

Nous réimplémentons également la fonction `changeEvent()` pour appeler `retranslateUi()` sur les événements `LanguageChange`. Qt génère un événement `LanguageChange` lorsque le contenu d'un `QTranslator` installé sur `QApplication` change. Dans notre application, ceci se produit lorsque nous appelons `load()` sur `appTranslator` ou `qtTranslator` depuis `MainWindow::switchLanguage()` ou `MainWindow::changeEvent()`.

Les événements `LanguageChange` ne doivent pas être confondus avec les événements `LocaleChange`. Ces derniers sont générés par le système et demandent à l'application de charger une nouvelle traduction. Les événements `LanguageChange` sont générés par Qt et demandent aux widgets de l'application de traduire toutes leurs chaînes.

Lorsque nous avons implémenté `MainWindow`, il ne nous a pas été nécessaire de répondre à `LanguageChange`. Nous avons simplement appelé `retranslateUi()` à chaque appel de `load()` sur un `QTranslator`.

```
void JournalView::retranslateUi()
{
   QStringList labels;
   labels << tr("Time") << tr("Priority") << tr("Phone Number")
          << tr("Subject");
   setHorizontalHeaderLabels(labels);
}
```

La fonction `retranslateUi()` met à jour les en-têtes de colonnes avec les textes nouvellement traduits, complétant ainsi la partie traduction du code d'un widget écrit à la main. Pour ce qui est des widgets et des boîtes de dialogue développés avec *Qt Designer*, l'outil `uic` génère automatiquement une fonction similaire à `retranslateUi()`. Celle-ci est automatiquement appelée en réponse aux événements `LanguageChange`.

Traduire les applications

La traduction d'une application Qt qui contient des appels à `tr()` est un processus en trois étapes :

1. Exécution de `lupdate` pour extraire toutes les chaînes visibles par l'utilisateur du code source de l'application.
2. Traduction de l'application au moyen de *Qt Linguist*.
3. Exécution de `lrelease` pour générer des fichiers binaires `.qm` que l'application peut charger au moyen de `QTranslator`.

La responsabilité des étapes 1 et 3 revient aux développeurs. L'étape 2 est gérée par les traducteurs. Ce cycle peut être répété aussi souvent que nécessaire pendant le développement et la durée de vie de l'application.

Nous allons, par exemple, vous montrer comment traduire l'application Spreadsheet du Chapitre 3. Cette application contient déjà des appels de `tr()` encadrant chaque chaîne visible par l'utilisateur.

Dans un premier temps, nous devons modifier le fichier `.pro` de l'application pour préciser quelles langues nous souhaitons prendre en charge. Si, par exemple, nous voulons prendre en charge l'allemand et le français en plus de l'anglais, nous ajoutons l'entrée `TRANSLATIONS` suivante à `spreadsheet.pro` :

```
TRANSLATIONS = spreadsheet_de.ts \
               spreadsheet_fr.ts
```

Ici, nous mentionnons deux fichiers de traduction : un pour l'allemand et un pour le français. Ces fichiers seront créés lors de la première exécution de `lupdate` et seront chargés à chacune de ses exécutions ultérieures.

Ces fichiers possèdent normalement une extension `.ts`. Ils se trouvent dans un format XML simple et ne sont pas aussi compacts que les fichiers binaires `.qm` compris par `QTranslator`. La tâche de conversion des fichiers `.ts` lisibles par l'homme en fichiers `.qm` efficaces pour la machine revient à `lrelease`. Pour les esprits curieux, `.ts` et `.qm` signifient fichier "translation source" et fichier "Qt message", respectivement.

En supposant que nous nous trouvions dans le répertoire contenant le code source de l'application Spreadsheet, nous pouvons exécuter `lupdate` sur `spreadsheet` à partir de la ligne de commande comme suit :

```
lupdate -verbose spreadsheet.pro
```

L'option –`verbose` invite `lupdate` à fournir plus de commentaires que d'habitude. Voici la sortie attendue :

```
Updating 'spreadsheet_de.ts'...
   Found 98 source texts (98 new and 0 already existing)
Updating 'spreadsheet_fr.ts'...
   Found 98 source texts (98 new and 0 already existing)
```

Chaque chaîne qui apparaît dans un appel de `tr()` dans le code source de l'application est stockée dans les fichiers `.ts`, avec une traduction vide. Les chaînes qui apparaissent dans les fichiers `.ui` de l'application sont également incluses.

L'outil `lupdate` suppose par défaut que les arguments de `tr()` sont des chaînes Latin-1. Si tel n'est pas le cas, nous devons ajouter une entrée `CODECFORTR` au fichier `.pro`. Par exemple :

```
CODECFORTR    = EUC-JP
```

Cette opération doit être réalisée en plus de l'appel à `QTextCodec::setCodecForTr()` depuis la fonction `main()` de l'application.

Les traductions sont alors ajoutées aux fichiers `spreadsheet_de.ts` et `spreadsheet_fr.ts` au moyen de *Qt Linguist*.

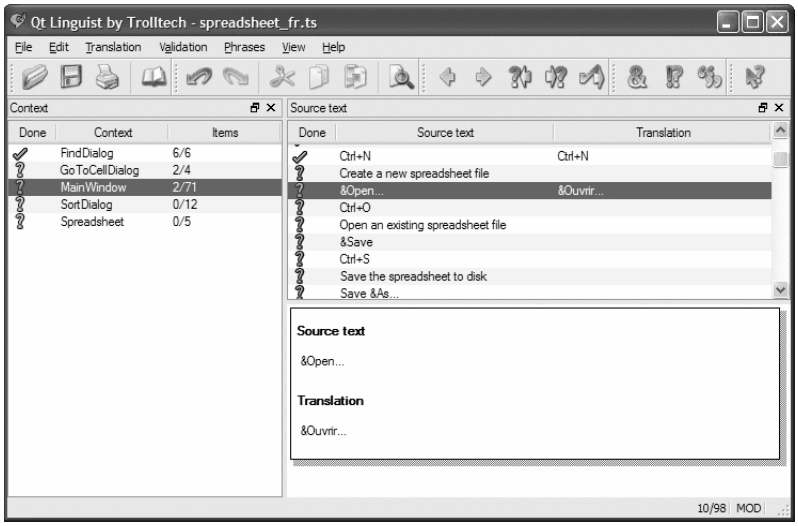
Pour exécuter *Qt Linguist,* cliquez sur Qt by Trolltech v4.x.y/Linguist dans le menu `Démarrer` sous Windows, saisissez `linguist` sur la ligne de commande sous Unix ou double-cliquez sur `Linguist` dans le Finder de Mac OS X. Pour commencer à ajouter des traductions à un fichier `.ts`, cliquez sur `File/Open` et sélectionnez le fichier à traduire.

La partie gauche de la fenêtre principale de *Qt Linguist* affiche la liste des contextes pour l'application en cours de traduction. En ce qui concerne l'application Spreadsheet, les contextes sont "FindDialog", "GoToCellDialog", "MainWindows", "SortDialog" et "Spreadsheet".La zone supérieure droite est la liste des textes sources pour le contexte en cours. Chaque texte source est présenté avec une traduction et un indicateur `Done`. La zone du milieu nous permet

d'entrer une traduction pour l'élément source activé. Dans la zone inférieure droite apparaît une liste de suggestions fournies automatiquement par *Qt Linguist*.

Lorsque nous disposons d'un fichier `.ts` traduit, nous devons le convertir en un fichier binaire `.qm` afin qu'il puisse être exploité par `QTranslator`. Pour effectuer cette opération depuis *Qt Linguist*, cliquez sur `File/Release`. Nous commençons généralement par traduire quelques chaînes et nous exécutons l'application avec le fichier `.qm` pour nous assurer que tout fonctionne correctement. (Voir Figure 17.2)

Figure 17.2
Qt Linguist en action



Si vous souhaitez générer les fichiers `.qm` pour tous les fichiers `.ts`, vous pouvez utiliser l'outil `lrelease` comme suit :

```
lrelease -verbose spreadsheet.pro
```

En supposant que vous avez traduit dix-neuf chaînes en français et cliqué sur l'indicateur `Done` pour dix-sept d'entre elles, `lrelease` produit la sortie suivante :

```
Updating 'spreadsheet_de.qm'...
   Generated 0 translations (0 finished and 0 unfinished)
   Ignored 98 untranslated source texts
Updating 'spreadsheet_fr.qm'...
   Generated 19 translations (17 finished and 2 unfinished)
   Ignored 79 untranslated source texts
```

Les chaînes non traduites sont présentées dans les langues initiales lors de l'exécution de l'application. L'indicateur `Done` est ignoré par `lrelease`. Il peut être utilisé par les traducteurs pour identifier les traductions terminées et celles qui ont encore besoin d'être révisées.

Lorsque nous modifions le code source de l'application, les fichiers de traduction deviennent obsolètes. La solution consiste à exécuter de nouveau `lupdate`, à fournir les traductions pour les nouvelles chaînes et à regénérer les fichiers `.qm`. Certaines équipes de développement considèrent qu'il faut régulièrement exécuter `lupdate`, alors que d'autres préfèrent attendre que l'application soit prête à être commercialisée.

Les outils `lupdate` et *Qt Linguist* sont relativement intelligents. Les traductions qui ne sont plus utilisées sont conservées dans les fichiers `.ts` au cas où elles seraient nécessaires dans des versions ultérieures. Lors de la mise à jour des fichiers `.ts`, `lupdate` utilise un algorithme de fusion intelligent qui permet aux traducteurs d'économiser un temps considérable dans le cas de texte identique ou similaire utilisé dans des contextes différents.

Pour plus d'informations concernant *Qt Linguist*, `lupdate` et `lrelease`, référez-vous au manuel *Qt Linguist* à l'adresse **http://doc.trolltech.com/4.1/linguist-manual.html**. Ce manuel contient une étude détaillée de l'interface utilisateur de *Qt Linguist* et un didactitiel pour les programmeurs.

18

Environnement multithread

Au sommaire de ce chapitre

- ✔ Créer des threads
- ✔ Synchroniser des threads
- ✔ Communiquer avec le thread principal
- ✔ Utiliser les classes Qt dans les threads secondaires

Les applications GUI conventionnelles possèdent un thread d'exécution et réalisent, en règle générale, une seule opération à la fois. Si l'utilisateur invoque une opération longue depuis l'interface utilisateur, cette dernière se fige pendant la progression de l'opération. Le Chapitre 7 présente des solutions à ce problème. L'environnement multithread en est une autre.

Dans une application multithread, l'interface utilisateur graphique s'exécute dans son propre thread et le traitement a lieu dans un ou plusieurs threads distincts. De cette façon, l'interface utilisateur graphique des applications reste réactive, même pendant un traitement intensif. Un autre avantage de l'environnement multitread est le suivant : les systèmes multiprocesseurs peuvent exécuter plusieurs threads simultanément sur différents processeurs, offrant ainsi de meilleures performances.

Dans ce chapitre, nous allons commencer par vous montrer comment dériver `QThread` et comment utiliser `QMutex`, `QSemaphore` ainsi que `QWaitCondition` pour synchroniser

les threads. Puis nous vous expliquerons comment communiquer avec le thread principal depuis les threads secondaires pendant que la boucle d'événement est en cours d'exécution. Nous conclurons enfin par une étude des classes Qt susceptibles ou non d'être utilisées dans des threads secondaires.

L'environnement multithread est un sujet vaste, auquel de nombreux livres sont exclusivement consacrés. Ici, nous supposons que vous maîtrisez les bases de la programmation multithread. Nous insisterons davantage sur la façon de développer des applications Qt multithread plutôt que sur le thème du découpage en threads lui-même.

Créer des threads

Il n'est pas difficile de fournir plusieurs threads dans une application Qt : il suffit de dériver `QThread` et de réimplémenter sa fonction `run()`. Pour illustrer cette opération, nous allons commencer par examiner le code d'une sous-classe `QThread` très simple qui affiche de façon répétée une chaîne donnée sur une console.

```
class Thread : public QThread
{
   Q_OBJECT
public:
   Thread();

   void setMessage(const QString &message);
   void stop();

protected:
   void run();

private:
   QString messageStr;
   volatile bool stopped;
};
```

La classe `Thread` hérite de `QThread` et réimplémente la fonction `run()`. Elle fournit deux fonctions supplémentaires : `setMessage()` et `stop()`.

La variable `stopped` est déclarée volatile car il est possible d'y accéder depuis plusieurs threads et nous souhaitons garantir une lecture de sa version actualisée à chaque fois qu'elle est nécessaire. Si nous omettons le mot-clé `volatile`, le compilateur peut optimiser l'accès à la variable, ce qui risque d'aboutir à des résultats incorrects.

```
Thread::Thread()
{
   stopped = false;
}
```

Nous avons défini `stopped` en `false` dans le constructeur.

```
void Thread::run()
{
   while (!stopped)
      cerr << qPrintable(messageStr);
   stopped = false;
   cerr << endl;
}
```

La fonction `run()` est appelée pour lancer l'exécution du thread. Tant que la variable `stopped` est définie en `false`, la fonction continue à afficher le message donné sur la console. Le thread se termine lorsque la fonction `run()` rend le contrôle.

```
void Thread::stop()
{
   stopped = true;
}
```

La fonction `stop()` définit la variable `stopped` en `true`, indiquant ainsi à `run()` d'interrompre l'affichage du texte sur la console. Cette fonction peut être appelée à tout moment depuis un thread quelconque. Dans le cadre de cet exemple, nous supposons que l'affectation à un `bool` est une opération atomique. C'est une supposition raisonnable, si l'on considère qu'un `bool` ne peut avoir que deux états. Nous verrons ultérieurement comment utiliser `QMutex` pour garantir l'atomicité de l'affectation à une variable.

`QThread` fournit une fonction `terminate()` qui met fin à l'exécution d'un thread pendant qu'il est toujours en cours d'exécution. L'emploi de `terminate()` n'est pas conseillé, car elle peut mettre fin au thread à tout moment et ne donne à ce dernier aucune chance de lancer les opérations de récupération de la mémoire. Il est toujours préférable de faire appel à une variable `stopped` et à une fonction `stop()` comme c'est le cas ici.

Figure 18.1
L'application Threads

A présent, nous allons voir comment utiliser la classe `Thread` dans une petite application Qt qui utilise deux threads, A et B, en plus du thread principal.

```
class ThreadDialog : public QDialog
{
   Q_OBJECT

public:
   ThreadDialog(QWidget *parent = 0);

protected:
   void closeEvent(QCloseEvent *event);

private slots:
   void startOrStopThreadA();
   void startOrStopThreadB();
```

```
private:
   Thread threadA;
   Thread threadB;
   QPushButton *threadAButton;
   QPushButton *threadBButton;
   QPushButton *quitButton;
};
```

La classe `ThreadDialog` déclare deux variables de type `Thread` et quelques boutons pour fournir une interface utilisateur de base.

```
ThreadDialog::ThreadDialog(QWidget *parent)
   : QDialog(parent)
{
   threadA.setMessage("A");
   threadB.setMessage("B");

   threadAButton = new QPushButton(tr("Start A"));
   threadBButton = new QPushButton(tr("Start B"));
   quitButton = new QPushButton(tr("Quit"));
   quitButton->setDefault(true);

   connect(threadAButton, SIGNAL(clicked()),
           this, SLOT(startOrStopThreadA()));

   connect(threadBButton, SIGNAL(clicked()),
           this, SLOT(startOrStopThreadB()));
   ...
}
```

Dans le constructeur, nous appelons `setMessage()` pour amener le premier et le second thread à afficher de façon répétée des 'A' et des 'B', respectivement.

```
void ThreadDialog::startOrStopThreadA()
{
   if (threadA.isRunning()) {
       threadA.stop();
       threadAButton->setText(tr("Start A"));
   } else {
       threadA.start();
       threadAButton->setText(tr("Stop A"));
   }
}
```

Lorsque l'utilisateur clique sur le bouton correspondant au thread A, `startOrStopThreadA()` stoppe ce dernier s'il était en cours d'exécution ou le lance dans le cas contraire. Elle met également à jour le texte du bouton.

```
void ThreadDialog::startOrStopThreadB()
{
   if (threadB.isRunning()) {
```

```
        threadB.stop();
        threadBButton->setText(tr("Start B"));
    } else {
        threadB.start();
        threadBButton->setText(tr("Stop B"));
    }
}
```

Le code de `startOrStopThreadB()` est très similaire.

```
void ThreadDialog::closeEvent(QCloseEvent *event)
{
    threadA.stop();
    threadB.stop();
    threadA.wait();
    threadB.wait();
    event->accept();
}
```

Si l'utilisateur clique sur `Quit` ou ferme la fenêtre, nous arrêtons tous les threads en cours d'exécution et attendons leur fin (en exécutant `QThread::wait()`) avant d'appeler `QCloseEvent::accept()`. Ce processus assure une fermeture correcte de l'application, bien qu'il ne présente pas vraiment d'importance dans le cadre de cet exemple.

Si vous exécutez l'application et cliquez sur `Start A`, la console se remplit de 'A'. Si vous cliquez sur `Start B`, elle se remplit maintenant de séquences alternatives de 'A' et de 'B'. Cliquez sur `Stop A`, et elle n'affiche alors que les 'B'.

Synchroniser des threads

La synchronisation de plusieurs threads représente une nécessité pour la plupart des applications multithread. Qt fournit les classes de synchronisation suivantes : `QMutex`, `QReadWriteLock`, `QSemaphore` et `QWaitCondition`.

La classe `QMutex` offre un moyen de protéger une variable ou une partie de code de sorte qu'un seul thread puisse y accéder à la fois. Elle fournit une fonction `lock()` qui verrouille le mutex. Si ce dernier est déverrouillé, le thread en cours s'en empare immédiatement et le verrouille. Dans le cas contraire, le thread en cours est bloqué jusqu'à ce que celui qui le contient le déverrouille. Dans tous les cas, lorsque l'appel à `lock()` se termine, le thread en cours conserve le mutex jusqu'à ce que `unlock()` soit appelé. La classe `QMutex` fournit également une fonction `tryLock()` qui se termine immédiatement si le mutex est déjà verrouillé.

Supposons, par exemple que nous souhaitions protéger la variable `stopped` de la classe `Thread` de la section précédente avec un `QMutex`. Nous ajouterions alors la donnée membre suivante à `Thread` :

```
private:
    ...
```

```
    QMutex mutex;
};
```

La fonction `run()` prendrait la forme suivante :

```
void Thread::run()
{
    forever {
        mutex.lock();
        if (stopped) {
            stopped = false;
            mutex.unlock();
            break;
        }
        mutex.unlock();

        cerr << qPrintable(messageStr);
    }
    cerr << endl;
}
```

La fonction `stop()` prendrait la forme suivante :

```
void Thread::stop()
{
    mutex.lock();
    stopped = true;
    mutex.unlock();
}
```

Les opérations de verrouillage et de déverrouillage d'un mutex dans les fonctions complexes, ou celles utilisant des exceptions C++, peuvent être sujettes à erreur. Qt offre la classe utilitaire `QMutexLocker` pour simplifier la gestion des mutex. Le constructeur de `QMutexLocker` accepte `QMutex` comme argument et le verrouille. Le destructeur de `QMutexLocker` déverrouille le mutex. Nous pourrions, par exemple, réécrire les fonctions `run()` et `stop()` comme suit :

```
void Thread::run()
{
    forever {
        {
            QMutexLocker locker(&mutex);
            if (stopped) {
                stopped = false;
                break;
            }
        }
        cerr << qPrintable(messageStr);
    }
    cerr << endl;
}
```

```
void Thread::stop()
{
   QMutexLocker locker(&mutex);
   stopped = true;
}
```

Le problème de l'emploi des mutex est que seul un thread peut accéder à la même variable à la fois. Dans les programmes avec de nombreux threads essayant de lire la même variable simultanément (sans la modifier), le mutex risque de compromettre sérieusement les performances. Dans ces situations, nous pouvons faire appel à `QReadWriteLock`, une classe de synchronisation qui autorise des accès simultanés en lecture seulement sans altérer les performances.

Dans la classe `Thread`, il serait illogique de remplacer `QMutex` par `QReadWriteLock` pour protéger la variable `stopped`, car un thread au maximum tentera de lire la variable à un moment donné. Un exemple plus approprié impliquerait un ou plusieurs threads de lecture accédant à des données partagées et un ou plusieurs threads d'écriture modifiant les données. Par exemple :

```
MyData data;
QReadWriteLock lock;

void ReaderThread::run()
{

   ...
   lock.lockForRead();
   access_data_without_modifying_it(&data);
   lock.unlock();
   ...
}

void WriterThread::run()
{

   ...
   lock.lockForWrite();
   modify_data(&data);
   lock.unlock();
   ...
}
```

Pour des raisons de commodité, il est possible d'utiliser les classes `QReadLocker` et `QWriteLocker` pour verrouiller et déverrouiller un `QReadWriteLock`.

`QSemaphore` est une autre généralisation des mutex, mais contrairement au verrouillage en lecture/écriture, les sémaphores peuvent être employés pour protéger un certain nombre de ressources identiques. Les deux extraits de code suivants montrent la correspondance entre `QSemaphore` et `QMutex` :

```
QSemaphore semaphore(1); QMutex mutex;
semaphore.acquire(); mutex.lock();
semaphore.release(); mutex.unlock();
```

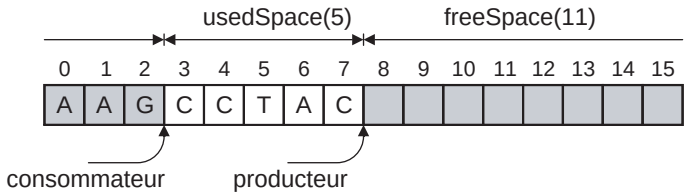
En transmettant `1` au constructeur, nous indiquons au sémaphore qu'il ne contrôle qu'une seule ressource. L'avantage de l'emploi d'un sémaphore est que nous pouvons transmettre un nombre supérieur à 1 au constructeur, puis appeler `acquire()` à plusieurs reprise pour acquérir de nombreuses ressources.

Une application typique des sémaphores est le transfert d'une certaine quantité de données (`DataSize`) entre deux threads à l'aide d'une mémoire tampon circulaire partagé d'une certaine taille (`BufferSize`) :

```
const int DataSize = 100000;
const int BufferSize = 4096;
char buffer[BufferSize];
```

Le thread producteur écrit les données dans la mémoire tampon jusqu'à ce qu'il atteigne la fin, puis recommence au début, écrasant les données existantes. Le thread consommateur lit les données lors de sa génération. La Figure 18.2 illustre ce processus, avec une mémoire tampon minuscule de 16 octets.

Figure 18.2
Le modèle producteur/consommateur



Le besoin de synchronisation dans l'exemple producteur/consommateur est double : si le producteur génère les données trop rapidement, il écrasera celles que le consommateur n'aura pas encore lues et si le consommateur lit trop rapidement, il dépassera le producteur et obtiendra des données incohérentes.

Un moyen radical de résoudre ce problème consiste à amener le producteur à remplir la mémoire tampon, puis à attendre que le consommateur ait lu cette dernière en entier, et ainsi de suite. Mais sur les machines multiprocesseurs, ce processus n'est pas aussi rapide que celui consistant à laisser les threads producteur et consommateur agir sur différentes parties de la mémoire tampon en même temps.

Pour résoudre efficacement ce problème, il est possible de recourir à deux sémaphores :

```
QSemaphore freeSpace(BufferSize);
QSemaphore usedSpace(0);
```

Le sémaphore `freeSpace` gouverne la partie de la mémoire tampon que le producteur remplit de données. Le sémaphore `usedSpace` gouverne la zone susceptible d'être lue par le consommateur. Ces deux régions sont complémentaires. Le sémaphore `freeSpace` est initialisé avec `BufferSize` (4096), ce qui signifie qu'il possède autant de ressources qu'il est possible d'en acquérir. Lorsque l'application démarre, le thread lecteur commence par acquérir des octets

"libres" et les convertit en octets "utilisés".Le sémaphore `usedSpace` est initialisé avec la valeur `0` pour s'assurer que le consommateur ne lit pas des données incohérentes au démarrage.

Dans cet exemple, chaque octet est considéré comme une ressource. Dans une application du monde réel, nous agirions probablement sur des unités de taille plus importante (64 ou 256 octets à la fois, par exemple) pour réduire la surcharge associée à l'emploi des sémaphores.

```
void Producer::run()
{
   for (int i = 0; i < DataSize; ++i) {
      freeSpace.acquire();
      buffer[i % BufferSize] = "ACGT"[uint(rand()) % 4];
      usedSpace.release();
   }
```

Dans le producteur, chaque itération débute par l'acquisition d'un octet "libre".Si la mémoire tampon est remplie de données n'ayant pas encore été lues par le consommateur, l'appel à `acquire()` bloquera le processus jusqu'à ce que le consommateur ait commencé à consommer les données. Une fois l'octet acquis, nous le remplissons de données aléatoires ('A', 'C', 'G' ou 'T') et le libérons avec le statut "utilisé", de sorte qu'il puisse être lu par le thread consommateur.

```
void Consumer::run()
{
   for (int i = 0; i < DataSize; ++i) {
      usedSpace.acquire();
      cerr << buffer[i % BufferSize];
      freeSpace.release();
   }
   cerr << endl;
}
```

Dans le consommateur, nous commençons par acquérir un octet "utilisé".Si la mémoire tampon ne contient aucune donnée à lire, l'appel à `acquire()` bloquera le processus jusqu'à ce que le producteur en ait produites. Une fois l'octet acquis, nous l'affichons et le libérons avec le statut "libre", en permettant ainsi au producteur de le remplir de nouveau avec des données.

```
int main()
{
   Producer producer;
   Consumer consumer;
   producer.start();
   consumer.start();
   producer.wait();
   consumer.wait();
   return 0;
}
```

Enfin, nous lançons les threads producteur et consommateur dans `main()`. Le producteur convertit alors de l'espace "libre" en espace "utilisé", puis le consommateur le reconvertit en espace "libre".

Lorsque nous exécutons le programme, il écrit une séquence aléatoire de 100 000 'A', 'C', 'G' et 'T' sur la console et se termine. Pour vraiment comprendre ce qui se passe, nous pouvons désactiver l'écriture de la sortie et écrire à la place un 'P' à chaque fois que le producteur génère un octet et 'c' à chaque fois que le consommateur en lit un. Pour simplifier les choses, nous pouvons utiliser des valeurs plus petites pour `DataSize` et `BufferSize`.

Voici, par exemple, une exécution possible avec un `DataSize` de `10` et un `BufferSize` de `4` : "PcPcPcPcPcPcPcPcPcPc". Dans ce cas, le consommateur lit les octets dès qu'ils sont générés par le producteur. Les deux threads sont exécutés à la même vitesse. Il est également possible que le producteur remplisse toute la mémoire tampon avant que le consommateur ne commence sa lecture : "PPPPccccPPPPccccPPcc". Il existe de nombreuses autres possibilités. Les sémaphores offrent une grande liberté au planificateur de thread spécifique au système, qui peut étudier le comportement des threads et choisir une stratégie de planification appropriée.

Une approche différente au problème de synchronisation d'un producteur et d'un consommateur consiste à employer `QWaitCondition` et `QMutex`. Un `QWaitCondition` permet à un thread d'en réveiller d'autres lorsque les conditions requises sont remplies. Cette approche autorise un contrôle plus précis qu'avec les mutex seuls. Pour illustrer le fonctionnement de ce processus, nous allons reprendre l'exemple producteur/consommateur en déclarant des conditions d'attente.

```
const int DataSize = 100000;
const int BufferSize = 4096;
char buffer[BufferSize];

QWaitCondition bufferIsNotFull;
QWaitCondition bufferIsNotEmpty;
QMutex mutex;
int usedSpace = 0;
```

En plus du tampon, nous déclarons deux `QWaitConditions`, un `QMutex` et une variable qui stocke le nombre d'octets "utilisés" dans la mémoire tampon.

```
void Producer::run()
{
   for (int i = 0; i < DataSize; ++i) {
      mutex.lock();
      while (usedSpace == BufferSize)
         bufferIsNotFull.wait(&mutex);
      buffer[i % BufferSize] = "ACGT"[uint(rand()) % 4];
      ++usedSpace;
      bufferIsNotEmpty.wakeAll();
      mutex.unlock();
   }
}
```

Dans le producteur, nous commençons par vérifier si la mémoire tampon est remplie. Si tel est le cas, nous attendons la condition "buffer is not full".Une fois cette condition remplie, nous écrivons un octet dans la mémoire tampon, nous incrémentons `usedSpace`, puis nous réveillons tout thread en attente de la condition "buffer is not full".

Nous faisons appel à un mutex pour protéger tous les accès à la variable `usedSpace`. La fonction `QWaitCondition::wait()` peut recevoir un mutex verrouillé comme premier argument, qu'elle déverrouille avant de bloquer le thread en cours puis verrouille avant de rendre le contrôle.

Dans cet exemple, nous aurions pu remplacer la boucle `while`

```
while (usedSpace == BufferSize)
   bufferIsNotFull.wait(&mutex);
```

par cette instruction `if` :

```
if (usedSpace == BufferSize) {
   mutex.unlock();
   bufferIsNotFull.wait();
   mutex.lock();
}
```

Ce processus risque d'être perturbé si nous autorisons plusieurs threads producteurs, car un autre producteur pourrait se saisir du mutex immédiatement après l'appel de `wait()` et annuler la condition "buffer is not full".

```
void Consumer::run()
{
   for (int i = 0; i < DataSize; ++i) {
      mutex.lock();
      while (usedSpace == 0)
         bufferIsNotEmpty.wait(&mutex);
      cerr << buffer[i % BufferSize];
      --usedSpace;
      bufferIsNotFull.wakeAll();
      mutex.unlock();
   }
   cerr << endl;
}
```

L'action du consommateur est très exactement opposée à celle du producteur : il attend la condition "buffer is not full" et réveille tout thread en attente de la vérification de cette condition.

Dans tous les exemples étudiés jusqu'à présent, nos threads ont accédé aux mêmes variables globales. Mais dans certaines applications, une variable globale doit contenir différentes valeurs dans différents threads. Ce procédé se nomme souvent stockage local de thread ou donnée spécifique aux threads. Nous pouvons le reproduire en utilisant un map indexé sur les ID de thread (retournés par `QThread::currentThread()`), mais une approche plus élégante consiste à faire appel à la classe `QThreadStorage<T>`.

Cette classe est généralement utilisée pour les caches. En ayant un cache distinct dans les différents threads, nous évitons la surcharge du verrouillage, déverrouillage et d'attente d'un mutex. Par exemple :

```
QThreadStorage<QHash<int, double> *> cache;

void insertIntoCache(int id, double value)
{
   if (!cache.hasLocalData())
       cache.setLocalData(new QHash<int, double>);
   cache.localData()->insert(id, value);
}

void removeFromCache(int id)
{
   if (cache.hasLocalData())
       cache.localData()->remove(id);
}
```

La variable `cache` contient un pointeur vers un `QMap<int,double>` par thread. Lorsque nous utilisons le cache pour la première fois dans un thread particulier, `hasLocalData()` retourne `false` et nous créons l'objet `QHash<int,double>`.

`QThreadStorage<T>` peut également être utilisé pour les variables d'état d'erreur globales (exactement comme `errno`) afin de s'assurer que les modifications apportées à un thread n'affectent pas les autres.

Communiquer avec le thread principal

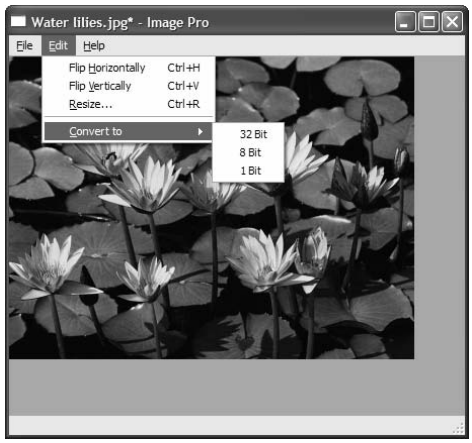
Quand une application Qt démarre, un seul thread s'exécute – le thread principal. C'est le seul thread qui est autorisé à créer l'objet `QApplication` ou `QCoreApplication` et à appeler `exec()` sur celui-ci. Après l'appel à `exec()`, ce thread est soit en attente d'un événement, soit en cours de traitement d'un événement.

Le thread principal peut démarrer de nouveaux threads en créant les objets d'une sous-classe `QThread`, comme ce fut le cas dans la section précédente. Si ces nouveaux threads doivent communiquer entre eux, ils peuvent utiliser des variables partagées avec des mutex, des verrouillages en lecture/écriture, des sémaphores ou des conditions d'attente. Mais aucune de ces techniques ne peut être exploitée pour communiquer avec le thread principal, car elles verrouilleraient la boucle de l'événement et figeraient l'interface utilisateur.

La solution pour communiquer avec le thread principal depuis un thread secondaire consiste à faire appel à des connexions signal/slot entre threads. En règle générale, le mécanisme des signaux et des slots opère de façon synchrone. Les slots sont donc connectés à un signal et invoqués immédiatement une fois le signal émis, par le biais d'un appel de fonction direct.

Cependant, lorsque nous connectons des objets "actifs" dans des threads différents, le mécanisme devient asynchrone. (Ce comportement peut être modifié par le biais d'un cinquième paramètre facultatif de `QObject::connect()`.) Ces connexions sont implémentées à l'arrière-plan en publiant un événement. Le slot est alors appelé par la boucle d'événement du thread dans laquelle se trouve l'objet destinataire. Par défaut, un `QObject` se situe dans le thread où il a été créé. Ceci peut être changé à tout moment en appelant `QObject::moveToThread()`.

Figure 18.3
L'application Image Pro



Pour illustrer le fonctionnement des connexions signal/slot entre threads, nous allons étudier le code de l'application Image Pro, une application de retouche d'images de base qui permet à l'utilisateur de faire pivoter, de redimensionner et de changer la précision des couleurs d'une image. L'application utilise un thread secondaire pour pouvoir réaliser des opérations sur les images sans verrouiller la boucle d'événement. La différence est significative lors du traitement de très grosses images. Le thread secondaire possède une liste de tâches, ou "transactions", à accomplir et envoie les événements à la fenêtre principale pour indiquer la progression.

```
ImageWindow::ImageWindow()
{
   imageLabel = new QLabel;
   imageLabel->setBackgroundRole(QPalette::Dark);
   imageLabel->setAutoFillBackground(true);
   imageLabel->setAlignment(Qt::AlignLeft | Qt::AlignTop);
   setCentralWidget(imageLabel);

   createActions();
   createMenus();

   statusBar()->showMessage(tr("Ready"), 2000);

   connect(&thread, SIGNAL(transactionStarted(const QString &)),
           statusBar(), SLOT(showMessage(const QString &)));
   connect(&thread, SIGNAL(finished()),
```

```
            this, SLOT(allTransactionsDone()));
    setCurrentFile("");
}
```

La partie la plus intéressante du constructeur de `ImageWindow` contient les deux connexions signal/slot. Toutes deux impliquent des signaux émis par l'objet `TransactionThread`, sur lequel nous reviendrons dans un instant.

```
void ImageWindow::flipHorizontally()
{
    addTransaction(new FlipTransaction(Qt::Horizontal));
}
```

Le slot `flipHorizontally()` crée une transaction "de rotation" et l'enregistre au moyen de la fonction privée `addTransaction()`. Les fonctions `flipVertically()`, `resizeImage()`, `convertTo32Bit()`, `convertTo8Bit()` et `convertTo1Bit()` sont similaires.

```
void ImageWindow::addTransaction(Transaction *transact)
{
    thread.addTransaction(transact);
    openAction->setEnabled(false);
    saveAction->setEnabled(false);
    saveAsAction->setEnabled(false);
}
```

La fonction `addTransaction()` ajoute une transaction à la file d'attente de transactions du thread secondaire et désactive les actions `Open`, `Save` et `Save As` pendant que les transactions sont en cours de traitement.

```
void ImageWindow::allTransactionsDone()
{
    openAction->setEnabled(true);
    saveAction->setEnabled(true);
    saveAsAction->setEnabled(true);
    imageLabel->setPixmap(QPixmap::fromImage(thread.image()));
    setWindowModified(true);
    statusBar()->showMessage(tr("Ready"), 2000);
}
```

Le slot `allTransactionsDone()` est appelé quand la file d'attente de transactions de `TransactionThread` se vide.

Passons maintenant à la classe `TransactionThread` :

```
class TransactionThread : public QThread
{
    Q_OBJECT

public:
    void addTransaction(Transaction *transact);
    void setImage(const QImage &image);
    QImage image();
```

```
signals:
   void transactionStarted(const QString &message);

protected:
   void run();

private:
   QMutex mutex;
   QImage currentImage;
   QQueue<Transaction *> transactions;
};
```

La classe `TransactionThread` conserve une liste des transactions à gérer et les exécute les unes après les autres à l'arrière-plan.

```
void TransactionThread::addTransaction(Transaction *transact)
{
   QMutexLocker locker(&mutex);
   transactions.enqueue(transact);
   if (!isRunning())
      start();
}
```

La fonction `addTransaction()` ajoute une transaction à la file d'attente des transactions et lance le thread de cette transaction s'il n'est pas déjà en cours d'exécution. Tous les accès à la variable membre `transactions` sont protégés par un mutex, le thread principal risquant de les modifier par l'intermédiaire de `addTransaction()` pendant que le thread secondaire parcourt `transactions`.

```
void TransactionThread::setImage(const QImage &image)
{
   QMutexLocker locker(&mutex);
   currentImage = image;
}

QImage TransactionThread::image()
{
   QMutexLocker locker(&mutex);
   return currentImage;
}
```

Les fonctions `setImage()` et `image()` permettent au thread principal de définir l'image sur laquelle les transactions doivent être effectuées et de récupérer l'image résultante une fois toutes les transactions terminées. Nous protégeons de nouveau les accès à une variable membre en utilisant un mutex.

```
void TransactionThread::run()
{
   Transaction *transact;
```

```
    forever {
        mutex.lock();
        if (transactions.isEmpty()) {
            mutex.unlock();
            break;
        }
        QImage oldImage = currentImage;
        transact = transactions.dequeue();
        mutex.unlock();

        emit transactionStarted(transact->message());

        QImage newImage = transact->apply(oldImage);
        delete transact;

        mutex.lock();
        currentImage = newImage;
        mutex.unlock();
    }
}
```

La fonction `run()` parcourt la file d'attente des transactions et les exécute chacune tour à tour en appelant `apply()` sur celles-ci.

Quand une transaction est lancée, nous émettons le signal `transactionStarted()` avec un message à afficher dans la barre d'état de l'application. Une fois le traitement de toutes les transactions terminé, la fonction `run()` se termine et `QThread` émet le signal `finished()`.

```
class Transaction
{
public:
    virtual ~Transaction() { }

    virtual QImage apply(const QImage &image) = 0;
    virtual QString message() = 0;
};
```

La classe `Transaction` est une classe de base abstraite pour les opérations susceptibles d'être effectuées par l'utilisateur sur une image. Le destructeur virtuel est nécessaire, car nous devons supprimer les instances des sous-classes `Transaction` par le biais d'un pointeur `Transaction`. (Si nous l'omettons, certains compilateurs émettent un avertissement.) `Transaction` possède trois sous-classes concrètes : `FlipTransaction`, `ResizeTransaction` et `ConvertDepthTransaction`. Nous n'étudierons que `FlipTransaction`, les deux autres classes étant similaires.

```
class FlipTransaction : public Transaction
{
public:
    FlipTransaction(Qt::Orientation orientation);

    QImage apply(const QImage &image);
    QString message();
```

```
private:
   Qt::Orientation orientation;
};
```

Le constructeur `FlipTransaction` reçoit un paramètre qui précise l'orientation de la rotation (horizontale ou verticale).

```
QImage FlipTransaction::apply(const QImage &image)
{
   return image.mirrored(orientation == Qt::Horizontal,
                         orientation == Qt::Vertical);
}
```

La fonction `apply()` appelle `QImage::mirrored()` sur le `QImage` reçu en tant que paramètre et retourne le `QImage` résultant.

```
QString FlipTransaction::message()
{
   if (orientation == Qt::Horizontal) {
       return QObject::tr("Flipping image horizontally...");
   } else {
       return QObject::tr("Flipping image vertically...");
   }
}
```

La fonction `message()` retourne le message à afficher dans la barre d'état pendant la progression de l'opération. Cette fonction est appelée dans `TransactionThread::run()` lors de l'émission du signal `transactionStarted()`.

Utiliser les classes Qt dans les threads secondaires

Une fonction est dite *thread-safe* quand elle peut être appelée sans problème depuis plusieurs threads simultanément. Si deux fonctions thread-safe sont appelées depuis des threads différents sur la même donnée partagée, le résultat est toujours défini. Par extension, une classe est dite thread-safe quand toutes ses fonctions peuvent être appelées simultanément depuis des threads différents sans qu'il y ait d'interférences entre elles.

Les classes thread-safe de Qt sont `QMutex`, `QMutexLocker`, `QReadWriteLock`, `QReadLocker`, `QWriteLocker`, `QSemaphore`, `QThreadStorage<T>`, `QWaitCondition` et certaines parties de l'API de `QThread`. En outre, plusieurs fonctions sont thread-safe, dont `QObject::connect()`, `QObject::disconnect()`, `QCoreApplication::postEvent()`, `QCoreApplication::removePostedEvent()` et `QCoreApplication::removePostedEvents()`.

La plupart des classes non GUI de Qt répondent à une exigence moins stricte : elles sont *réentrantes*. Une classe est réentrante si différentes instances de la classe peuvent être utilisées simultanément dans des threads différents. Cependant, l'accès simultané au même objet réentrant dans plusieurs threads n'est pas sécurisé, et un tel accès doit être protégé avec un mutex. Les classes

réentrantes sont marquées comme telles dans la documentation de référence Qt. En règle générale, toute classe C++ ne faisant pas référence à une donnée globale ou partagée est réentrante.

`QObject` est réentrant, mais trois contraintes doivent être prises en considération :

- Les `QObject` enfants doivent être créés dans le thread de leur parent.

  Ceci signifie que les objets créés dans un thread secondaire ne doivent jamais être créés avec le même objet `QThread` que leur parent, car cet objet a été créé dans un autre thread (soit dans le thread principal, soit dans un thread secondaire différent).
- Nous devons supprimer tous les `QObject` créés dans un thread secondaire avant d'envisager la suppression de l'objet `QThread` correspondant.

  Pour ce faire, nous pouvons créer les objets sur la pile dans `QThread::run()`.
- Les `QObject` doivent être supprimés dans le thread les ayant créés.

  Si nous devons supprimer un `QObject` situé dans un thread différent, nous devons appeler la fonction `QObject::deleteLater()`, qui publie un événement "deferred delete" (suppression différée).

Les sous-classes `QObject` non GUI, telles que `QTimer`, `QProcess` et les classes de réseau, sont réentrantes. Nous pouvons les utiliser dans tout thread, ceci tant que ce thread possède une boucle d'événement. En ce qui concerne les threads secondaires, la boucle d'événement est démarrée en appelant `QThread::exec()` ou par des fonctions utilitaires telles que `QProcess::waitForFinished()` et `QAbstractSocket::waitForDisconnected()`.

`QWidget` et ses sous-classes ne sont pas réentrantes, à cause des limites héritées des bibliothèques sur lesquelles repose la prise en charge GUI de Qt. Par conséquent, nous ne pouvons pas appeler directement des fonctions depuis un thread secondaire sur un widget. Pour changer, par exemple, le texte d'un `QLabel` depuis un thread secondaire, nous pouvons émettre un signal connecté à `QLabel::setText()` ou appeler `QMetaObject::invokeMethod()` depuis ce thread. Par exemple :

```
void MyThread::run()
{
   ...
   QMetaObject::invokeMethod(label, SLOT(setText(const QString &)),
                             Q_ARG(QString, "Hello"));
   ...
}
```

De nombreuses classes non GUI de Qt, dont `QImage`, `QString` et les classes conteneur, utilisent le partage implicite comme technique d'optimisation. Bien que cette optimisation ne rende généralement pas une classe réentrante, ceci ne pose pas de problème dans Qt. En effet, des instructions de langage assembleur atomiques sont employées pour implémenter un décompte de références thread-safe, qui rend les classes implicitement partagées de Qt réentrantes.

Le module *QtSql* de Qt peut également être employé dans les applications multithread, mais il présente des restrictions qui varient d'une base de données à une autre. Vous trouverez plus de détails à l'adresse **http://doc.trolltech.com/4.1/sql-driver.html**.

19

Créer des plug-in

Au sommaire de ce chapitre

- ✔ Développer Qt avec les plug-in
- ✔ Créer des applications capables de gérer les plug-in
- ✔ Ecrire des plug-in pour des applications

Les bibliothèques dynamiques (également nommées bibliothèques partagées ou DLL) sont des modules indépendants stockés dans un fichier séparé sur le disque et auxquels de multiples applications peuvent accéder. Les programmes spécifient généralement les bibliothèques dynamiques qui leur sont nécessaires au moment de la liaison, auquel cas ces bibliothèques sont automatiquement chargées lors du démarrage de l'application. Cette approche implique généralement l'ajout de la bibliothèque ainsi que de son chemin d'accès d'inclusion au fichier `.pro` de l'application. En outre, les en-têtes adéquats sont inclus dans les fichiers sources. Par exemple :

```
LIBS          += -ldb_cxx
INCLUDEPATH   += /usr/local/BerkeleyDB.4.2/include
```

L'alternative consiste à charger dynamiquement la bibliothèque quand elle est requise, puis à résoudre les symboles que nous souhaitons utiliser. Qt fournit la classe `QLibrary` pour y parvenir d'une manière indépendante de la plate-forme. A partir d'un nom de bibliothèque,

`QLibrary` examine les emplacements standard sur la plate-forme en question à la recherche d'un fichier approprié. Pour le nom `mimetype`, par exemple, elle recherchera `mimetype.dll` sous Windows, `mimetype.so` sous Linux et `mimetype.dylib` sous Mac OS X.

Les applications GUI modernes peuvent souvent être développées par l'emploi des plug-in. Un plug-in est une bibliothèque dynamique qui implémente une interface particulière pour fournir une fonctionnalité supplémentaire facultative. Par exemple, dans le Chapitre 5, nous avons créé un plug-in pour intégrer un widget personnalisé à *Qt Designer*.

Qt reconnaît son propre ensemble d'interfaces de plugin pour divers domaines, dont les formats d'image, les pilotes de bases de données, les styles de widgets, le codage du texte et l'accessibilité. La première section de ce chapitre explique comment développer Qt avec un plug-in.

Il est également possible de créer des plug-in spécifiques à des applications Qt particulières. Qt facilite l'écriture de tels éléments par le biais de sa structure de plug-in, qui sécurise également `QLibrary` contre les pannes et rend son utilisation plus aisée. Dans les deux dernières sections de ce chapitre, nous vous montrerons comment créer une application qui prend en charge les plug-in et comment générer un plug-in personnalisé pour une application.

Développer Qt avec les plug-in

Qt peut être développé avec plusieurs types de plug-in, les plus courants étant les pilotes de base de données, les formats d'images, les styles et les codecs de texte. Pour chaque type de plug-in, nous avons généralement besoin d'un minimum de deux classes : une classe wrapper (conteneur) de plug-in qui implémente les fonctions API génériques, et une ou plusieurs classes gestionnaires qui implémentent chacune l'API d'un type de plug-in particulier. Les gestionnaires sont accessibles par le biais de la classe wrapper. (Voir Figure 19.1).

Classe de plug-in	*Classe de base gestionnaire*
`QAccessibleBridgePlugin`	`QAccessibleBridge`
`QAccessiblePlugin`	`QAccessibleInterface`
`QIconEnginePlugin`	`QIconEngine`
`QImageIOPlugin`	`QImageIOHandler`
`QInputContextPlugin`	`QInputContext`
`QPictureFormatPlugin`	`N/A`
`QSqlDriverPlugin`	`QSqlDriver`
`QStylePlugin`	`QStyle`
`QTextCodecPlugin`	`QTextCodec`

Figure 19.1
Les classes gestionnaire et de plug-in de Qt (à l'exception de `Qtopia Core`*)*

Pour illustrer ceci, nous allons implémenter un plug-in capable de lire des fichiers curseur (fichiers `.cur`) Windows monochromes. Ces fichiers peuvent contenir plusieurs images du même curseur à des tailles différentes. Une fois le plug-in de curseur créé et installé, Qt sera en mesure de lire les fichiers `.cur` et d'accéder aux curseurs individuels (par le biais de `QImage`, `QImageReader` ou `QMovie`, par exemple). Il pourra également convertir les curseurs en tout autre format de fichier d'image, tel que BMP, JPEG et PNG. Le plug-in pourrait aussi être déployé avec les applications Qt, car elles contrôlent automatiquement les emplacements standard des plug-in Qt et chargent ceux qu'elles trouvent.

Les nouveaux conteneurs de plug-in de format d'image doivent sous-classer `QImageIOPlugin` et réimplémenter quelques fonctions virtuelles :

```
class CursorPlugin : public QImageIOPlugin
{
public:
   QStringList keys() const;
   Capabilities capabilities(QIODevice *device,
                             const QByteArray &format) const;
   QImageIOHandler *create(QIODevice *device,
                           const QByteArray &format) const;
};
```

La fonction `keys()` retourne une liste de formats d'image pris en charge par le plug-in. Le paramètre `format` des fonctions `capabilities()` et `create()` est supposé avoir une valeur qui provient de cette liste.

```
QStringList CursorPlugin::keys() const
{
   return QStringList() << "cur";
}
```

Notre plug-in ne prend en charge qu'un seul format d'image. Il retourne donc une liste avec un nom unique. Idéalement, le nom doit être l'extension de fichier utilisée pour ce format. Dans le cas de formats avec plusieurs extensions (telles que `.jpg` et `.jpeg` pour `JPEG`), nous pouvons retourner une liste avec plusieurs entrées, chacune correspondant à une extension.

```
QImageIOPlugin::Capabilities
CursorPlugin::capabilities(QIODevice *device,
                           const QByteArray &format) const
{
   if (format == "cur")
       return CanRead;

   if (format.isEmpty()) {
       CursorHandler handler;
       handler.setDevice(device);
       if (handler.canRead())
           return CanRead;
   }

   return 0;
}
```

La fonction `capabilities()` retourne ce que le gestionnaire d'image est en mesure d'accomplir avec le format d'image donné. Il existe trois possibilités (`CanRead`, `CanWrite` et `CanReadIncremental`), et la valeur de retour est un opérateur de bits `OR` des possibilités qui s'appliquent.

Si le format est "cur", notre implémentation retourne `CanRead`. Si aucun format n'est fourni, nous créons un gestionnaire de curseur et vérifions s'il est capable de lire les données depuis le périphérique en question. La fonction `canRead()` lit uniquement les données, essayant de déterminer si elle reconnaît le fichier, sans en changer le pointeur. Une possibilité de `0` indique que ce gestionnaire ne peut ni lire ce fichier, ni y écrire des données.

```
QImageIOHandler *CursorPlugin::create(QIODevice *device,
                                      const QByteArray &format) const
{
   CursorHandler *handler = new CursorHandler;
   handler->setDevice(device);
   handler->setFormat(format);
   return handler;
}
```

Quand un fichier curseur est ouvert (par `QImageReader`, par exemple), la fonction `create()` du conteneur de plug-in est appelée avec le pointeur de périphérique et avec le format "cur". Nous créons une instance de `CursorHandler` et la configurons avec le périphérique et le format spécifiés. L'appelant prend la propriété du gestionnaire et le supprimera quand il ne sera plus utile. Si plusieurs fichiers doivent être lus, un nouveau gestionnaire peut être créé pour chacun.

```
Q_EXPORT_PLUGIN2(cursorplugin, CursorPlugin)
```

A la fin du fichier `.cpp`, nous utilisons la macro `Q_EXPORT_PLUGIN2()` pour s'assurer que le plug-in est reconnu par Qt. Le premier paramètre est un nom arbitraire que nous souhaitons attribuer au plug-in. Le second paramètre est le nom de classe du plug-in.

Il est facile de sous-classer `QImageIOPlugin`. Le travail véritable du plug-in est effectué dans le gestionnaire. Les gestionnaires de format d'image doivent sous-classer `QImageIOHandler` et réimplémenter ses fonctions publiques en totalité ou en partie. Commençons par l'en-tête :

```
class CursorHandler : public QImageIOHandler
{
public:
   CursorHandler();

   bool canRead() const;
   bool read(QImage *image);
   bool jumpToNextImage();
   int currentImageNumber() const;
   int imageCount() const;

private:
   enum State { BeforeHeader, BeforeImage, AfterLastImage, Error };

   void readHeaderIfNecessary() const;
```

```
    QBitArray readBitmap(int width, int height, QDataStream &in) const;
    void enterErrorState() const;

    mutable State state;
    mutable int currentImageNo;
    mutable int numImages;
};
```

Les signatures de toutes les fonctions publiques sont fixes. Nous avons omis quelques fonctions dont la réimplémentation est inutile pour un gestionnaire en lecture seulement, en particulier `write()`. Les variables de membre sont déclarées avec le mot-clé `mutable`, car elles sont modifiées à l'intérieur des fonctions `const`.

```
CursorHandler::CursorHandler()
{
    state = BeforeHeader;
    currentImageNo = 0;
    numImages = 0;
}
```

Lors de la construction du gestionnaire, nous commençons par définir son état. Nous appliquons au premier curseur le numéro d'image du curseur en cours. Ici, `numImage` étant défini en `0`, il est évident que nous n'avons pas encore d'image.

```
bool CursorHandler::canRead() const
{
    if (state == BeforeHeader) {
        return device()->peek(4) == QByteArray("\0\0\2\0", 4);
    } else {
        return state != Error;
    }
}
```

La fonction `canRead()` peut être appelée à tout moment pour déterminer si le gestionnaire d'images peut lire des données supplémentaires depuis le périphérique. Si la fonction est appelée avant que nous ayons lu toute donnée et alors que nous nous trouvons toujours dans l'état `BeforeHeader`, nous vérifions la signature particulière qui identifie les fichiers curseur Windows. L'appel `QIODevice::peek()` lit les quatre premiers octets *sans* changer le pointeur du fichier du périphérique. Si `canRead()` est appelée ultérieurement, nous retournons `true` quand aucune erreur ne s'est produite.

```
int CursorHandler::currentImageNumber() const
{
    return currentImageNo;
}
```

Cette fonction retourne le numéro du curseur au niveau duquel le pointeur de fichier du périphérique est positionné. Une fois le gestionnaire construit, il est possible pour l'utilisateur d'appeler toutes ses fonctions publiques, dans un ordre quelconque. C'est un problème potentiel, car nous devons partir du principe que nous ne pouvons effectuer qu'une lecture séquentielle.

Nous devons donc lire l'en-tête de fichier au moins une fois avant tout autre élément. Nous résolvons le problème en appelant la fonction `readHeaderIfNecessary()`.

```
int CursorHandler::imageCount() const
{
   readHeaderIfNecessary();
   return numImages;
}
```

Cette fonction retourne le nombre d'images dans le fichier. Pour un fichier valide où aucune erreur de lecture ne s'est produite, elle retournera un décompte d'au moins 1 (voir Figure 19.2).

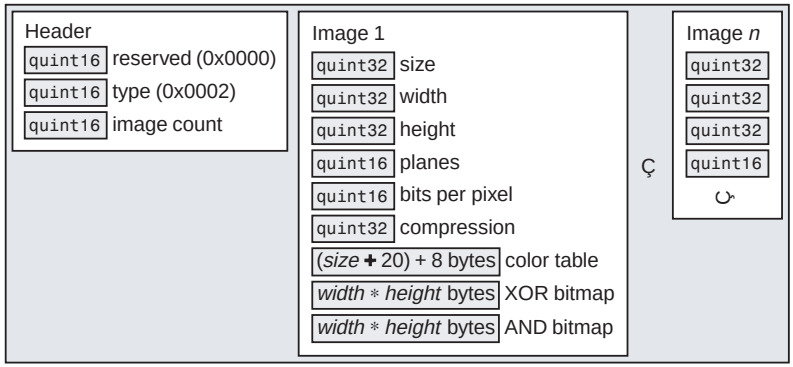


Figure 19.2
Le format de fichier `.cur`

La fonction suivante est importante. Nous l'étudierons donc par fragments :

```
bool CursorHandler::read(QImage *image)
{
   readHeaderIfNecessary();

   if (state != BeforeImage)
       return false;
```

La fonction `read()` lit les données de toute image, en commençant à l'emplacement du pointeur de périphérique courant. Nous pouvons lire l'image suivante si l'en-tête du fichier est lu avec succès, ou après qu'une image ait été lue et que le pointeur de périphérique se déplace au début d'une autre image.

```
quint32 size;
quint32 width;
quint32 height;
quint16 numPlanes;
quint16 bitsPerPixel;
quint32 compression;

QDataStream in(device());
```

```
in.setByteOrder(QDataStream::LittleEndian);
in >> size;
if (size != 40) {
    enterErrorState();
    return false;
}
in >> width >> height >> numPlanes >> bitsPerPixel >> compression;
height /= 2;

if (numPlanes != 1 || bitsPerPixel != 1 || compression != 0) {
    enterErrorState();
    return false;
}

in.skipRawData((size - 20) + 8);
```

Nous créons un `QDataStream` pour lire le périphérique. Nous devons définir un ordre d'octets correspondant à celui mentionné par la spécification de format de fichier `.cur`. Il n'est pas besoin de définir un numéro de version `QDataStream`, car le format de nombres entiers et de nombres à virgule flottante ne varie pas entre les versions de flux de données. Nous lisons ensuite les divers éléments de l'en-tête du curseur. Nous passons les parties inutiles de l'en-tête et la table des couleurs de 8 octets au moyen de `QDataStream::skipRawData()`. Nous devons revoir toutes les caractéristiques du format – par exemple en diminuant de moitié la hauteur car le format `.cur` fournit une hauteur qui est deux fois supérieure à celle de l'image véritable. Les valeurs `bitsPerPixel` et `compression` sont toujours de `1` et de `0` dans un fichier `.cur` monochrome. Si nous rencontrons des problèmes, nous pouvons appeler `enterErrorState()` et retourner `false`.

```
QBitArray xorBitmap = readBitmap(width, height, in);
QBitArray andBitmap = readBitmap(width, height, in);
if (in.status() != QDataStream::Ok) {
    enterErrorState();
    return false;
}
```

Les éléments suivants du fichier sont deux bitmaps, l'une étant un masque `XOR` et l'autre un masque `AND`. Nous les lisons dans les `QBitArray` plutôt que dans les `QBitmap`. Un `QBitmap` est une classe conçue pour être dessinée à l'écran, mais ici, nous avons besoin d'un tableau d'octets ordinaire. Lorsque nous en avons fini avec la lecture du fichier, nous vérifions l'état du `QDataStream`. Cette opération fonctionne correctement, car si un `QDataStream` entre dans un état d'erreur, il y reste et ne peut retourner que des zéros. Si, par exemple, la lecture échoue au niveau du premier tableau d'octets, la tentative de lecture du deuxième résulte en un `QBitArray` vide.

```
*image = QImage(width, height, QImage::Format_ARGB32);

for (int i = 0; i < int(height); ++i) {
   for (int j = 0; j < int(width); ++j) {
      QRgb color;
```

```
            int bit = (i * width) + j;

            if (andBitmap.testBit(bit)) {
               if (xorBitmap.testBit(bit)) {
                   color = 0x7F7F7F7F;
               } else {
                  color = 0x00FFFFFF;
               }
            } else {
               if (xorBitmap.testBit(bit)) {
                   color = 0xFFFFFFFF;
            } else {
                   color = 0xFF000000;
               }
            }
            image->setPixel(j, i, color);
          }
       }
```

Nous construisons un nouveau `QImage` de la taille correcte et définissons `image` de façon qu'il pointe ver celui-ci. Puis nous parcourons chaque pixel des tableaux d'octets `XOR` et `AND` et nous les convertissons en spécifications de couleur ARGB de 32 bits. Les tableaux d'octets `AND` et `XOR` sont utilisés comme présenté dans le tableau suivant pour obtenir la couleur de chaque pixel du curseur :

AND	*XOR*	*Résultat*
1	1	Pixel d'arrière-plan inversé
1	0	Pixel transparent
0	1	Pixel blanc
0	0	Pixel noir

Les pixels noirs, blancs et transparents ne présentent pas de problème, mais il n'existe aucun moyen d'obtenir un pixel d'arrière-plan inversé au moyen d'une spécification de couleur ARGB sans connaître la couleur du pixel d'arrière-plan initial. En remplacement, nous utilisons une couleur grise semi-transparente (`0x7F7F7F7F`).

```
        ++currentImageNo;
        if (currentImageNo == numImages)
            state = AfterLastImage;
        return true;
    }
```

Une fois la lecture de l'image terminée, nous mettons à jour le numéro de l'image courante ainsi que l'état si nous avons atteint la dernière image. A la fin de la fonction, le périphérique sera positionné au niveau de l'image suivante ou à la fin du fichier.

```
bool CursorHandler::jumpToNextImage()
{
   QImage image;
   return read(&image);
}
```

La fonction `jumpToNextImage()` permet de passer une image. Pour des raisons de simplicité, nous appelons simplement `read()` et ignorons le `QImage` résultant. Une implémentation plus efficace consisterait à utiliser l'information stockée dans l'en-tête du fichier `.cur` pour passer directement à l'offset approprié dans le fichier.

```
void CursorHandler::readHeaderIfNecessary() const
{
   if (state != BeforeHeader)
      return;

   quint16 reserved;
   quint16 type;
   quint16 count;

   QDataStream in(device());
   in.setByteOrder(QDataStream::LittleEndian);

   in >> reserved >> type >> count;
   in.skipRawData(16 * count);

   if (in.status() != QDataStream::Ok || reserved != 0
         || type != 2 || count == 0) {
       enterErrorState();
       return;
   }

   state = BeforeImage;
   currentImageNo = 0;
   numImages = int(count);
}
```

La fonction privée `readHeaderIfNecessary()` est appelée depuis `imageCount()` et `read()`. Si l'en-tête du fichier a déjà été lu, l'état n'est pas `BeforeHeader` et nous rendons immédiatement le contrôle. Dans le cas contraire, nous ouvrons un flux de données sur le périphérique, lisons quelques données génériques (dont le nombre de curseurs dans le fichier) et définissons l'état en `BeforeImage`. Le pointeur de fichier du périphérique est positionné avant la première image.

```
void CursorHandler::enterErrorState() const
{
   state = Error;
   currentImageNo = 0;
   numImages = 0;
}
```

Si une erreur se produit, nous supposons qu'il n'existe pas d'images valides et définissons l'état en Error. L'état du gestionnaire ne peut alors plus être modifié.

```
QBitArray CursorHandler::readBitmap(int width, int height,
                                    QDataStream &in) const
{
   QBitArray bitmap(width * height);
   quint8 byte;
   quint32 word;

   for (int i = 0; i < height; ++i) {
       for (int j = 0; j < width; ++j) {
           if ((j % 32) == 0) {
              word = 0;
              for (int k = 0; k < 4; ++k) {
                  in >> byte;
                  word = (word << 8) | byte;
              }
           }

           bitmap.setBit(((height - i - 1) * width) + j,
                         word & 0x80000000);
           word <<= 1;
      }
   }
   return bitmap;
}
```

La fonction `readBitmap()` permet de lire les masques `AND` et `XOR` d'un curseur. Ces masques ont deux fonctions inhabituelles. En premier lieu, ils stockent les lignes de bas en haut, au lieu de l'approche courante de haut en bas. En second lieu, l'ordre des données apparaît inversé par rapport à celui utilisé à tout autre emplacement dans les fichiers `.cur`. Nous devons donc inverser la coordonnée *y* dans l'appel de `setBit()`, et nous lisons les valeurs bit par bit dans le masque.

Nous terminons ici l'implémentation du plug-in de format d'image `CursorHandler`. Les plug-in correspondant à d'autres formats d'image doivent suivre le même schéma, bien que certains puissent implémenter une part plus importante de l'API `QImageIOHandler`, en particulier les fonctions utilisées pour écrire les images. Les plug-in d'autres types, tels que les codecs de texte ou les pilotes de base de données, suivent le même schéma : un conteneur de plug-in fournit une API générique que les applications peuvent utiliser, ainsi qu'un gestionnaire fournissant les fonctionnalités sous-jacentes.

Le fichier `.pro` diffère entre les plug-in et les applications. Nous terminerons avec ceci :

```
TEMPLATE      = lib
CONFIG       += plugin
HEADERS       = cursorhandler.h \
                cursorplugin.h
SOURCES       = cursorhandler.cpp \
                cursorplugin.cpp
DESTDIR       = $(QTDIR)/plugins/imageformats
```

Par défaut, les fichiers `.pro` font appel au modèle `app`, mais ici, nous devons utiliser le modèle `lib` car un plug-in est une bibliothèque, et non une application autonome. La ligne `CONFIG` indique à Qt que la bibliothèque n'est pas classique, mais qu'il s'agit d'une bibliothèque de plug-in. `DESTDIR` mentionne le répertoire où doit être placé le plug-in. Tous les plug-in Qt doivent se trouver dans le sous-répertoire `plugins` approprié où a été installé Qt, et comme notre plug-in fournit un nouveau format d'image, nous le plaçons dans le sous-répertoire `plugins/imageformats`. La liste des noms de répertoire et des types de plug-in est fournie à l'adresse **http://doc.trolltech.com/4.1/plugins-howto.html**. Pour cet exemple, nous supposons que la variable d'environnement `QTDIR` est définie dans le répertoire où est installé Qt.

Les plug-in créés pour Qt en mode release et en mode debug sont différents. Ainsi, si les deux versions de Qt sont installées, il est conseillé de préciser laquelle doit être utilisée dans le fichier `.pro`, en ajoutant par exemple la ligne suivante :

```
CONFIG    += release
```

Les applications qui utilisent les plug-in Qt doivent être déployées avec ceux qu'elles sont sensées exploiter. Les plug-in Qt doivent être placés dans des sous-répertoires spécifiques (`imageformats`, par exemple, pour les formats d'image). Les applications Qt recherchent les plug-in dans le sous-répertoire `plugins` situé dans le dossier où réside leur exécutable. Ainsi, pour les plug-in d'image elles recherchent dans `application_dir/plugins/imageformats`. Si nous souhaitons déployer les plug-in Qt dans un répertoire différent, les directions de recherche des plug-in peuvent être multipliées en utilisant `QCoreApplication::addLibraryPath()`.

Créer des applications capables de gérer les plug-in

Un plug-in d'application est une bibliothèque dynamique qui implémente une ou plusieurs *interfaces*. Une interface est une classe qui est constituée exclusivement de fonctions purement virtuelles. La communication entre l'application et les plug-in s'effectue par le biais de la table virtuelle de l'interface. Dans cette section, nous nous pencherons sur l'emploi d'un plug-in dans une application Qt par l'intermédiaire de ses interfaces, et dans la section suivante nous verrons comment implémenter un plug-in.

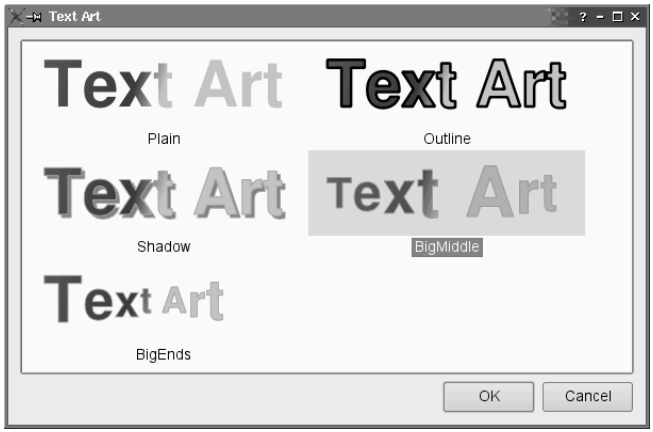
Pour fournir un exemple concret, nous allons créer l'application Text Art simple présentée en Figure 19.3. Les effets de texte sont fournis par les plug-in. L'application récupère la liste des effets de texte fournis par chaque plug-in et les parcourt pour afficher chacun d'eux en tant qu'élément dans un `QListWidget` (voir Figure 19.3).

L'application Text Art définit une interface :

```
class TextArtInterface
{
```

```
public:
   virtual ~TextArtInterface() { }

   virtual QStringList effects() const = 0;
   virtual QPixmap applyEffect(const QString &effect,
                               const QString &text,
                               const QFont &font, const QSize &size,
                               const QPen &pen,
                               const QBrush &brush) = 0;
};

Q_DECLARE_INTERFACE(TextArtInterface,
                    "com.software-inc.TextArt.TextArtInterface/1.0")
```

Figure 19.3
L'application Text Art



Une classe d'interface déclare normalement un destructeur virtuel, une fonction virtuelle qui retourne un `QStringList` ainsi qu'une ou plusieurs autres fonctions virtuelles. Le destructeur est avant tout destiné à satisfaire le compilateur, qui, sans sa présence se plaindrait du manque de destructeur virtuel dans une classe possédant des fonctions virtuelles. Dans cet exemple, la fonction `effects()` retourne une liste d'effets de texte fournis par le plug-in. Nous pouvons envisager cette liste comme une liste de clés. A chaque fois que nous appelons une autre fonction, nous transmettons l'une de ces clés en tant que premier argument, permettant ainsi l'implémentation d'effets multiples dans un plug-in.

Nous faisons finalement appel à la macro `Q_DECLARE_INTERFACE()` pour associer un identificateur à l'interface. Cet identificateur comprend généralement quatre composants : un nom de domaine inverse spécifiant le créateur de l'interface, le nom de l'application, le nom de l'interface et un numéro de version. Dès que nous modifions l'interface (en ajoutant une fonction virtuelle ou en changeant la signature d'une fonction existante), nous devons incrémenter le numéro de version. Dans le cas contraire, l'application risque de tomber brusquement en panne en essayant d'accéder à un plug-in obsolète.

L'application est implémentée dans une classe nommée `TextArtDialog`. Nous n'étudierons ici que le code présentant un intérêt. Commençons par le constructeur :

```
TextArtDialog::TextArtDialog(const QString &text, QWidget *parent)
   : QDialog(parent)
{
   listWidget = new QListWidget;
   listWidget->setViewMode(QListWidget::IconMode);
   listWidget->setMovement(QListWidget::Static);
   listWidget->setIconSize(QSize(260, 80));
   ...
   loadPlugins();
   populateListWidget(text);
   ...
}
```

Le constructeur crée un `QListWidget` pour répertorier les effets disponibles. Il appelle la fonction privée `loadPlugins()` pour rechercher et charger tout plug-in implémentant le `TextArtInterface` et remplit le `QListWidget` en conséquence, en appelant une autre fonction privée, `populateListWidget()`.

```
void TextArtDialog::loadPlugins()
{
   QDir pluginDir(QApplication::applicationDirPath());

#if defined(Q_OS_WIN)
   if (pluginDir.dirName().toLower() == "debug"
           || pluginDir.dirName().toLower() == "release")
       pluginDir.cdUp();
#elif defined(Q_OS_MAC)
   if (pluginDir.dirName() == "Mac OS") {
       pluginDir.cdUp();
       pluginDir.cdUp();
       pluginDir.cdUp();
   }
#endif
   if (!pluginDir.cd("plugins"))
       return;

   foreach (QString fileName, pluginDir.entryList(QDir::Files)) {
       QPluginLoader loader(pluginDir.absoluteFilePath(fileName));
       if (TextArtInterface *interface =
                   qobject_cast<TextArtInterface *>(loader.instance()))
           interfaces.append(interface);
   }
}
```

Dans `loadPlugins()`, nous tentons de charger tous les fichiers du répertoire `plugins` de l'application. (Sous Windows, l'exécutable de l'application réside généralement dans un sous-répertoire `debug` ou `release`. Nous remontons donc d'un niveau. Sous Mac OS X, nous prenons en compte la structure de répertoire du paquet.)

Si le fichier que nous essayons de charger est un plug-in qui utilise la même version de Qt que l'application, `QPluginLoader::instance()` retournera un `QObject*` qui pointe vers un plug-in Qt. Nous exécutons `qobject_cast<T>()` pour vérifier si le plug-in implémente le `TextArtInterface`. A chaque fois que la conversion est réussie, nous ajoutons l'interface à la liste d'interfaces (de type `QList<TextArtInterface*>`) de `TextArtDialog`.

Certaines applications peuvent vouloir charger plusieurs interfaces différentes, auquel cas le code permettant d'obtenir les interfaces doit être similaire à celui-ci :

```
QObject *plugin = loader.instance();
if (TextArtInterface *i = qobject_cast<TextArtInterface *>(plugin))
    textArtInterfaces.append(i);
if (BorderArtInterface *i = qobject_cast<BorderArtInterface *>(plugin))
    borderArtInterfaces.append(i);

if (TextureInterface *i = qobject_cast<TextureInterface *>(plugin))
    textureInterfaces.append(i);
```

Les mêmes plug-in peuvent être convertis avec succès en plusieurs pointeurs d'interface, car il est possible pour les plug-in de fournir plusieurs interfaces avec l'héritage multiple.

```
void TextArtDialog::populateListWidget(const QString &text)
{
   QSize iconSize = listWidget->iconSize();
   QPen pen(QColor("darkseagreen"));

   QLinearGradient gradient(0, 0, iconSize.width() / 2,
                            iconSize.height() / 2);
   gradient.setColorAt(0.0, QColor("darkolivegreen"));
   gradient.setColorAt(0.8, QColor("darkgreen"));
   gradient.setColorAt(1.0, QColor("lightgreen"));

   QFont font("Helvetica", iconSize.height(), QFont::Bold);

   foreach (TextArtInterface *interface, interfaces) {
      foreach (QString effect, interface->effects()) {
         QListWidgetItem *item = new QListWidgetItem(effect,
                                                     listWidget);
         QPixmap pixmap = interface->applyEffect(effect, text, font,
                                                 iconSize, pen,
                                                 gradient);
         item->setData(Qt::DecorationRole, pixmap);
      }
   }
   listWidget->setCurrentRow(0);
}
```

La fonction `populateListWidget()` commence en créant quelques variables à transmette à la fonction `applyEffect()`, et en particulier un stylet, un dégradé linéaire et une police. Elle parcourt ensuit le `TextArtInterface` ayant été trouvé par `loadPlugins()`. Pour tout effet

fourni par chaque interface, un nouveau `QListWidgetItem` est créé avec le nom de l'effet qu'il représente, et un `QPixmap` est généré au moyen de `applyEffect()`.

Dans cette section, nous avons vu comment charger des plug-in en appelant `loadPlugins()` dans le constructeur, et comment les exploiter dans `populateListWidget()`. Le code détermine élégamment combien il existe de plug-in fournissant `TextArtInterface`. En outre, des plug-in supplémentaires peuvent être ajoutés ultérieurement : à chaque nouvelle ouverture de l'application, celle-ci charge tous les plug-in fournissant les interfaces souhaitées. Il est ainsi possible de développer la fonctionnalité de l'application sans pour autant la modifier.

Ecrire des plug-in d'application

Un plug-in d'application est une sous-classe de `QObject` et des interfaces qu'il souhaite fournir. Le CD d'accompagnement de ce livre inclut deux plug-in pour l'application Text Art présentée dans la section précédente. Ils visent à démontrer que l'application gère correctement plusieurs plug-in.

Dans cet ouvrage, nous n'étudierons le code que de l'un d'entre eux, le plug-in Basic Effects. Nous supposerons que le code source du plug-in est situé dans un répertoire nommé `basiceffects-plugin` et que l'application Text Art se trouve dans un répertoire parallèle nommé `textart`. Voici la déclaration de la classe de plug-in :

```
class BasicEffectsPlugin : public QObject, public TextArtInterface
{
   Q_OBJECT
   Q_INTERFACES(TextArtInterface)

public:
   QStringList effects() const;
   QPixmap applyEffect(const QString &effect, const QString &text,
                       const QFont &font, const QSize &size,
                       const QPen &pen, const QBrush &brush);
};
```

Le plug-in n'implémente qu'une seule interface, `TextArtInterface`. En plus de `Q_OBJECT`, nous devons utiliser la macro `Q_INTERFACES()` pour chaque interface sous-classée afin de nous assurer d'une coopération sans problèmes entre `moc` et `qobject_cast<T>()`.

```
QStringList BasicEffectsPlugin::effects() const
{
   return QStringList() << "Plain" << "Outline" << "Shadow";
}
```

La fonction `effects()` retourne une liste d'effets de texte pris en charge par le plug-in. Celui-ci supportant trois effets, nous retournons simplement une liste contenant le nom de chacun d'eux.

La fonction `applyEffect()` fournit la fonctionnalité du plug-in et s'avère assez importante. C'est pourquoi nous l'étudierons par fragments.

```
QPixmap BasicEffectsPlugin::applyEffect(const QString &effect,
        const QString &text, const QFont &font, const QSize &size,
        const QPen &pen, const QBrush &brush)
{
QFont myFont = font;
QFontMetrics metrics(myFont);
while ((metrics.width(text) > size.width()
        || metrics.height() > size.height())
       && myFont.pointSize() > 9) {
   myFont.setPointSize(myFont.pointSize() - 1);
   metrics = QFontMetrics(myFont);
}
```

Nous souhaitons nous assurer que le texte donné s'adaptera si possible à la taille spécifiée. C'est la raison pour laquelle nous utilisons les métriques de la police afin de déterminer si texte est trop gros pour s'adapter. Si tel est le cas, nous exécutons une boucle où nous réduisons la taille en points jusqu'à parvenir à une dimension correcte, ou jusqu'à parvenir à 9 points, notre taille minimale fixée.

```
QPixmap pixmap(size);

QPainter painter(&pixmap);
painter.setFont(myFont);
painter.setPen(pen);
painter.setBrush(brush);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setRenderHint(QPainter::TextAntialiasing, true);
painter.setRenderHint(QPainter::SmoothPixmapTransform, true);
painter.eraseRect(pixmap.rect());
```

Nous créons un pixmap de la taille requise et un painter pour peindre sur le pixmap. Nous définissons aussi quelques conseils de rendu pour assurer les meilleurs résultats possible. L'appel à `eraseRect()` efface le pixmap avec la couleur d'arrière-plan.

```
if (effect == "Plain") {
   painter.setPen(Qt::NoPen);
} else if (effect == "Outline") {
   QPen pen(Qt::black);
   pen.setWidthF(2.5);
   painter.setPen(pen);
} else if (effect == "Shadow") {
   QPainterPath path;
   painter.setBrush(Qt::darkGray);
   path.addText(((size.width() - metrics.width(text)) / 2) + 3,
                (size.height() - metrics.descent()) + 3, myFont,
                text);
   painter.drawPath(path);
   painter.setBrush(brush);
}
```

Pour l'effet "Plain" aucun relief n'est requis. En ce qui concerne l'effet "Outline", nous ignorons le stylet initial et en créons un de couleur noire avec une largeur de 2,5 pixels. Quant à l'effet "Shadow", il nous faut d'abord dessiner l'ombre, de façon à pouvoir inscrire le texte par-dessus.

```
QPainterPath path;
path.addText((size.width() - metrics.width(text)) / 2,
             size.height() - metrics.descent(), myFont, text);
   painter.drawPath(path);

   return pixmap;
}
```

A présent, nous disposons du stylet et de l'ensemble des pinceaux adaptés à chaque effet de texte. Nous sommes maintenant prêts à afficher le texte. Ce dernier est centré horizontalement et suffisamment éloigné du bas du pixmap pour laisser de la place aux hampes inférieures.

```
Q_EXPORT_PLUGIN2(basiceffectsplugin, BasicEffectsPlugin)
```

A la fin du fichier `.cpp`, nous exécutons la macro `Q_EXPORT_PLUGIN2()` afin de rendre le plug-in disponible pour Qt.

Le fichier `.pro` est similaire à celui que nous avons utilisé pour le plug-in de curseur Windows précédemment dans ce chapitre.

```
TEMPLATE        = lib
CONFIG         += plugin
HEADERS         = ../textart/textartinterface.h \<RC>basiceffectsplugin.h
SOURCES         = basiceffectsplugin.cpp
DESTDIR         = ../textart/plugins
```

Si ce chapitre vous a donné l'envie d'en savoir plus sur les plug-in, vous pouvez étudier l'exemple plus avancé Plug & Paint fourni avec Qt. L'application prend en charge trois interfaces différentes et inclut une boîte de dialogue Plugin Information répertoriant les plug-in et interfaces disponibles pour l'application.

20

Fonctionnalités spécifiques à la plate-forme

Au sommaire de ce chapitre

- ✔ Construire une interface avec les API natives
- ✔ ActiveX sous Windows
- ✔ Prendre en charge la gestion de session de X11

Dans ce chapitre, nous allons étudier quelques options spécifiques à la plate-forme disponibles pour les programmeurs Qt. Nous commencerons par examiner comment accéder aux API natives telles que l'API Win32 de Windows, Carbon sous Mac OS X et Xlib sous X11. Nous étudierons ensuite l'extension ActiveQt, pour apprendre à utiliser les contrôles ActiveX sous Windows. Vous découvrirez aussi que cette extension permet

de créer des applications qui se comportent comme des serveurs ActiveX. Dans la dernière section, nous expliquerons comment amener des applications Qt à coopérer avec le gestionnaire de session sous X11.

Pour compléter les fonctionnalités présentées ci-dessus, Trolltech offre plusieurs solutions Qt spécifiques à la plate-forme, notamment les frameworks de migration Qt/Motif et Qt/MFC qui simplifient la migration des applications Motif/Xt et MFC vers Qt. Une extension analogue est fournie pour les applications Tcl/Tk par *froglogic*, et un convertisseur de ressource Microsoft Windows est disponible auprès de Klarälvdalens Datakonsult. Consultez les pages Web suivantes pour obtenir des détails complémentaires :

- **`http://www.trolltech.com/products/solutions/catalog/`**
- **`http://www.froglogic.com/tq/`**
- **`http://www.kdab.net/knut/`**

Pour tout ce qui concerne le développement intégré, Trolltech propose la plate-forme d'application Qtopia. Cette plate-forme est détaillée au Chapitre 21.

Construire une interface avec les API natives

L'API très développée de Qt répond à la plupart des besoins sur toutes les plates-formes, mais dans certaines circonstances, vous pourriez avoir besoin d'employer les API locales. Dans cette section, nous allons montrer comment utiliser ces API pour les différentes plates-formes prises en charge par Qt afin d'accomplir des tâches particulières.

Figure 20.1
Une fenêtre d'outil Mac OS X avec la barre de titre sur le côté

Sur toutes les plates-formes, `QWidget` fournit une fonction `winId()` qui renvoie l'identifiant de la fenêtre ou le handle (descripteur). `QWidget` fournit aussi une fonction statique nommée `find()` qui renvoie le `QWidget` avec un ID de fenêtre particulier. Nous pouvons transmettre cet identifiant aux fonctions d'API natives pour obtenir des effets spécifiques à la plate-forme.

Par exemple, le code suivant s'appuie sur `winId()` pour déplacer la barre de titre d'une fenêtre d'outil vers la gauche à l'aide des fonctions natives de Mac OS X :

```
#ifdef Q_WS_MAC
   ChangeWindowAttributes(HIViewGetWindow(HIViewRef(toolWin.winId())),
                          kWindowSideTitlebarAttribute,
                          kWindowNoAttributes);
#endif
```

Sous X11, voici comment nous pourrions modifier une propriété de fenêtre :

```
#ifdef Q_WS_X11
   Atom atom = XInternAtom(QX11Info::display(), "MY_PROPERTY", False);
   long data = 1;
   XChangeProperty(QX11Info::display(), window->winId(), atom, atom,
                   32, PropModeReplace,
                   reinterpret_cast<uchar *>(&data), 1);
#endif
```

Les directives `#ifdef` et `#endif` qui encadrent le code spécifique à la plate-forme garantissent la compilation de l'application sur les autres plates-formes.

Pour une application uniquement destinée à Windows, voici un exemple d'utilisation des appels de GDI pour afficher un widget Qt :

```
void GdiControl::paintEvent(QPaintEvent * /* event */)
{
    RECT rect;
    GetClientRect(winId(), &rect);
    HDC hdc = GetDC(winId());
    FillRect(hdc, &rect, HBRUSH(COLOR_WINDOW + 1));
    SetTextAlign(hdc, TA_CENTER | TA_BASELINE);
    TextOutW(hdc, width() / 2, height() / 2, text.utf16(), text.size());
    ReleaseDC(winId(), hdc);
}
```

Pour que ce code s'exécute, nous devons aussi réimplémenter `QPaintDevice::paintEngine()` de sorte qu'elle renvoie un pointeur nul et définisse l'attribut `Qt::WA_PaintOnScreen` dans le constructeur du widget.

L'exemple suivant montre comment combiner `QPainter` et des appels GDI dans le gestionnaire d'événement paint en exécutant les fonctions `getDC()` et `releaseDC()` :

```
void MyWidget::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);
    painter.fillRect(rect().adjusted(20, 20, -20, -20), Qt::red);
#ifdef Q_WS_WIN
    HDC hdc = painter.paintEngine()->getDC();
    Rectangle(hdc, 40, 40, width() - 40, height() - 40);
    painter.paintEngine()->releaseDC();
#endif
}
```

Le mélange des appels de `QPainter` avec les appels GDI comme dans ce code conduit quelquefois à des résultats étranges, en particulier quand les appels de `QPainter` se produisent après les appels GDI. En effet, `QPainter` s'appuie dans ce cas sur certaines hypothèses concernant l'état de la couche de dessin sous-jacente.

Qt définit un des quatre symboles de système de fenêtre suivant : `Q_WS_WIN`, `Q_WS_X11`, `Q_WS_MAC`, et `Q_WS_QWS` (Qtopia). Nous devons inclure au moins un en-tête Qt pour être en mesure de les utiliser dans les applications. Qt fournit aussi des symboles de préprocesseur destinés à identifier le système d'exploitation :

- `Q_OS_AIX`
- `Q_OS_BSD4`
- `Q_OS_BSDI`
- `Q_OS_CYGWIN`
- `Q_OS_DGUX`
- `Q_OS_DYNIX`
- `Q_OS_FREEBSD`
- `Q_OS_HPUX`
- `Q_OS_HURD`
- `Q_OS_IRIX`
- `Q_OS_LINUX`
- `Q_OS_LYNX`
- `Q_OS_MAC`
- `Q_OS_NETBSD`
- `Q_OS_OPENBSD`
- `Q_OS_OS2EMX`
- `Q_OS_OSF`
- `Q_OS_QNX6`
- `Q_OS_QNX`
- `Q_OS_RELIANT`
- `Q_OS_SCO`
- `Q_OS_SOLARIS`
- `Q_OS_ULTRIX`
- `Q_OS_UNIXWARE`
- `Q_OS_WIN32`
- `Q_OS_WIN64`

Nous pouvons supposer qu'au moins un de ces symboles sera défini. Pour des raisons pratiques, Qt définit aussi `Q_OS_WIN` dès que Win32 ou Win64 est détecté, et `Q_OS_UNIX` lorsqu'un système d'exploitation de type UNIX (y compris Linux et Mac OS X), est reconnu. Au moment de l'exécution, nous pouvons contrôler `QSysInfo::WindowsVersion` ou `QSysInfo::MacintoshVersion` pour identifier les différentes versions de Windows (2000, XP, etc.) ou de Mac OS X (10.2, 10.3, etc.).

Des macros de compilateur viennent compléter les macros du système de fenêtrage et du système d'exploitation. Par exemple, `Q_CC_MSVC` est définie si le compilateur est Microsoft Visual C++. Elles peuvent se révéler très pratiques pour résoudre les erreurs de compilateur.

Plusieurs classes de Qt liées à l'interface graphique fournissent des fonctions spécifiques à la plate-forme qui renvoient des handles de bas niveau vers l'objet sous-jacent. Ces descripteurs sont énumérés en Figure 20.2.

Sous X11, `QPixmap::x11Info()` et `QWidget::x11Info()` renvoient un objet `QX11Info` qui fournit divers pointeurs ou handles, tels que `display()`, `screen()`, `colormap()` et `visual()`. Nous pouvons les utiliser pour configurer un contexte graphique X11 sur un `QPixmap` ou un `QWidget`, par exemple.

Les applications Qt qui doivent collaborer avec d'autres toolkits ou bibliothèques ont souvent besoin d'accéder aux événements de bas niveau sous X11, `MSGs` sous Windows, `EventRef` sous Mac OS X, `QWSEvents` sous Qtopia) avant qu'ils ne soient convertis en `QEvents`. Nous pouvons procéder en dérivant `QApplication` et en réimplémentant le filtre d'événement spécifique à la plate-forme approprié, c'est-à-dire `x11EventFilter()`, `winEventFilter()`,

Mac OS X	
`ATSFontFormatRef`	`QFont::handle()`
`CGImageRef`	`QPixmap::macCGHandle()`
`GWorldPtr`	`QPixmap::macQDAlphaHandle()`
`GWorldPtr`	`QPixmap::macQDHandle()`
`RgnHandle`	`QRegion::handle()`
`HIViewRef`	`QWidget::winId()`

Windows	
`HCURSOR`	`QCursor::handle()`
`HDC`	`QPaintEngine::getDC()`
`HDC`	`QPrintEngine::getPrinterDC()`
`HFONT`	`QFont::handle()`
`HPALETTE`	`QColormap::hPal()`
`HRGN`	`QRegion::handle()`
`HWND`	`QWidget::winId()`

X11	
`Cursor`	`QCursor::handle()`
`Font`	`QFont::handle()`
`Picture`	`QPixmap::x11PictureHandle()`
`Picture`	`QWidget::x11PictureHandle()`
`Pixmap`	`QPixmap::handle()`
`QX11Info`	`QPixmap::x11Info()`
`QX11Info`	`QWidget::x11Info()`
`Region`	`QRegion::handle()`
`Screen`	`QCursor::x11Screen()`
`SmcConn`	`QSessionManager::handle()`
`Window`	`QWidget::handle()`
`Window`	`QWidget::winId()`

Figure 20.2
Fonctions spécifiques à la plate-forme pour accéder aux handles de bas niveau

`macEventFilter()`, ou `qwsEventFilter()`. Nous pouvons aussi accéder aux événements de plate-forme qui sont envoyés à un widget donné en réimplémentant un des filtres `x11Event()`, `winEvent()`, `macEvent()`, et `qwsEvent()`. Cela pourrait présenter de l'intérêt pour la gestion de certains types d'événement qui seraient normalement ignorés dans QT, comme les événements de manette de jeu.

Vous trouverez des informations complémentaires concernant les détails spécifiques à chaque plate-forme, notamment comment déployer les applications Qt sur différentes plates-formes, à l'adresse **http://doc.trolltech.com/4.1/win-system.html**.

ActiveX sous Windows

La technologie ActiveX de Microsoft permet aux applications d'incorporer des composants d'interface utilisateur fournis par d'autres applications ou bibliothèques. Elle est basée sur Microsoft COM et définit un jeu d'interfaces pour application qui emploie des composants et un autre jeu d'interfaces pour application et bibliothèque qui fournit les composants.

La version Qt/Windows Desktop Edition fournit le Framework ActiveQt et combine de façon homogène ActiveX et Qt. ActiveQt est constitué de deux modules :

- Le module *QAxContainer* nous permet d'utiliser les objets COM et d'intégrer des contrôles ActiveX dans les applications Qt.
- Le module *QAxServer* nous permet d'exporter des objets COM personnalisés et des contrôles ActiveX écrits avec Qt.

Notre premier exemple va intégrer l'application Windows Media Player dans une application Qt à l'aide du module *QAxContainer*. L'application Qt ajoute un bouton `Open`, un bouton `Play/Pause`, un bouton `Stop` et un curseur au contrôle ActiveX de Windows Media Player.

Figure 20.3
L'application Media Player

La fenêtre principale de l'application est de type `PlayerWindow` :

```
class PlayerWindow : public QWidget

{
   Q_OBJECT
   Q_ENUMS(ReadyStateConstants)

public:
   enum PlayStateConstants { Stopped = 0, Paused = 1, Playing = 2 };
   enum ReadyStateConstants { Uninitialized = 0, Loading = 1,
                              Interactive = 3, Complete = 4 };

   PlayerWindow();

protected:
   void timerEvent(QTimerEvent *event);

private slots:
   void onPlayStateChange(int oldState, int newState);
   void onReadyStateChange(ReadyStateConstants readyState);
   void onPositionChange(double oldPos, double newPos);
   void sliderValueChanged(int newValue);
   void openFile();

private:
   QAxWidget *wmp;
   QToolButton *openButton;
   QToolButton *playPauseButton;
   QToolButton *stopButton;
   QSlider *seekSlider;
   QString fileFilters;
   int updateTimer;
};
```

La classe `PlayerWindow` hérite de `QWidget`. La macro `Q_ENUMS()` (directement sous `Q_OBJECT`) est indispensable pour signaler à moc que le type `ReadyStateConstants` utilisé dans le slot `onReadyStateChange()` est un type enum. Dans la section privée, nous déclarons la donnée membre `QAxWidget*`.

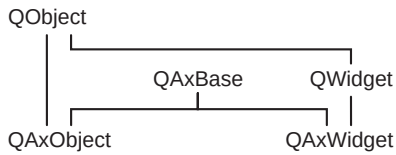
```
PlayerWindow::PlayerWindow()

{
    wmp = new QAxWidget;
    wmp->setControl("{22D6F312-B0F6-11D0-94AB-0080C74C7E95}");
```

Dans le constructeur, nous commençons par créer un objet `QAxWidget` afin d'encapsuler le contrôle ActiveX Windows Media Player. Le module `QAxContainer` se compose de trois classes : `QAxObject` englobe un objet COM, `QAxWidget` englobe un contrôle ActiveX, et `QAxBase` implémente la fonctionnalité COM pour `QAxObject` et `QAxWidget`.

Nous appelons `setControl()` sur `QAxWidget` avec l'ID de la classe du contrôle Windows Media Player 6.4. Une instance du composant requis va ainsi être créée. A partir de là, toutes

les propriétés, événements, et méthodes du contrôle ActiveX vont être disponibles sous la forme de propriétés, signaux et slots Qt, par le biais de l'objet `QAxWidget`.

Figure 20.4
Schéma d'héritage du module `QAxContainer`



Les types de données COM sont automatiquement convertis vers les types Qt correspondants, comme illustré en Figure 20.5. Par exemple, un paramètre d'entrée de type `VARIANT_BOOL` prend le type `bool`, et un paramètre de sortie de type `VARIANT_BOOL` devient un type `bool &`. Si le type obtenu est une classe Qt (`QString`, `QDateTime`, etc.), le paramètre d'entrée est une référence const (par exemple, `const QString &`).

Types COM	*Types Qt*
`VARIANT_BOOL`	`bool`
`char, short, int, long`	`int`
`unsigned char, unsigned short,`	`uint`
`unsigned int, unsigned long`	
`float, double`	`double`
`CY`	`qlonglong, qulonglong`
`BSTR`	`QString`
`DATE`	`QDateTime, QDate, QTime`
`OLE_COLOR`	`QColor`
`SAFEARRAY(VARIANT)`	`QList<QVariant>`
`SAFEARRAY(BSTR)`	`QStringList`
`SAFEARRAY(BYTE)`	`QByteArray`
`VARIANT`	`QVariant`
`IFontDisp *`	`QFont`
`IPictureDisp *`	`QPixmap`
`User defined type`	`QRect, QSize, QPoint`

Figure 20.5
Relations entre types COM et types Qt

Pour obtenir la liste de toutes les propriétés, signaux et slots disponibles pour un `QAxObject` ou un `QAxWidget` avec leurs types de données Qt, appelez `QAxBase::generateDocumentation()` ou faites appel à l'outil de ligne de commande de Qt `dumpdoc`, que vous trouverez dans le répertoire `tools\activeqt\dumpdoc` de Qt.

Examinons maintenant le constructeur de `PlayerWindow` :

```
    wmp->setProperty("ShowControls", false);
    wmp->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
    connect(wmp, SIGNAL(PlayStateChange(int, int)),
                        this, SLOT(onPlayStateChange(int, int)));
    connect(wmp, SIGNAL(ReadyStateChange(ReadyStateConstants)),
                        this, SLOT(onReadyStateChange(ReadyStateConstants)));
    connect(wmp, SIGNAL(PositionChange(double, double)),
                        this, SLOT(onPositionChange(double, double)));
```

Après l'appel de `QAxWidget::setControl()`, nous appelons `QObject::setProperty()` pour définir la propriété `ShowControls` du Windows Media Player en `false`, puisque nous fournissons nos propres boutons pour manipuler le composant. `QObject::setProperty()` peut être utilisée à la fois pour les propriétés COM et pour les propriétés Qt normales. Son second paramètre est de type `QVariant`.

Nous appelons ensuite `setSizePolicy()` de sorte que le contrôle ActiveX occupe toute la place disponible dans la disposition, et nous connectons trois événements ActiveX depuis le composant COM vers les trois slots.

```
    …
    stopButton = new QToolButton;
    stopButton->setText(tr("&Stop"));
    stopButton->setEnabled(false);
    connect(stopButton, SIGNAL(clicked()), wmp, SLOT(Stop()));
    …
}
```

La fin du constructeur `PlayerWindow` suit le modèle habituel, sauf que nous connectons quelques signaux Qt aux slots fournis par l'objet COM (`Play()`, `Pause()`, et `Stop()`). Les boutons étant analogues, nous ne présentons ici que l'implémentation du bouton `Stop`.

Il est temps d'aborder la fonction `timerEvent()` :

```
void PlayerWindow::timerEvent(QTimerEvent *event)
{

    if (event->timerId() == updateTimer) {
        double curPos = wmp->property("CurrentPosition").toDouble();
        onPositionChange(-1, curPos);
    } else {
        QWidget::timerEvent(event);
 }
}
```

La fonction `timerEvent()` est appelée à intervalles réguliers pendant la diffusion d'un clip. Nous l'employons pour faire avancer le curseur. Pour ce faire, nous appelons `property()` sur le contrôle ActiveX afin d'obtenir la valeur de la propriété `CurrentPosition` en tant que `QVariant` et nous appelons `toDouble()` pour le convertir en `double`. Nous appelons alors `onPositionChange()` pour réaliser la mise à jour.

La suite du code présente peu d'intérêt parce qu'elle ne concerne pas directement ActiveX et qu'elle ne comporte rien qui n'ait déjà été abordé. Le code complet est inclus sur le CD-ROM.

Dans le fichier `.pro`, nous devons introduire l'entrée suivante pour établir une liaison avec le module *QAxContainer* :

```
CONFIG     += qaxcontainer
```

Lorsqu'on gère des objets COM, on a souvent besoin d'appeler directement une méthode COM (et pas seulement de la connecter à un signal Qt). La méthode la plus simple de procéder est d'invoquer `QAxBase::dynamicCall()` avec le nom et la signature de la méthode en premier paramètre et les arguments de cette dernière comme paramètres supplémentaires. Saisissez par exemple :

```
wmp->dynamicCall("TitlePlay(uint)", 6);
```

La fonction `dynamicCall()` reçoit jusqu'à huit paramètres de type `QVariant` et renvoie un `QVariant`. Si nous avons besoin de transmettre un `IDispatch*` ou un `IUnknown*` de cette façon, nous pouvons encapsuler le composant dans un `QAxObject` et appeler `asVariant()` sur ce dernier pour le convertir en `QVariant`. Si nous avons besoin d'appeler une méthode COM qui renvoie un `IDispatch*` ou un `IUnknown*`, ou si nous avons besoin d'accéder à une propriété COM appartenant à l'un de ces types, nous pouvons opter plutôt pour `querySubObject()` :

```
QAxObject *session = outlook.querySubObject("Session");
QAxObject *defaultContacts =
      session->querySubObject("GetDefaultFolder(OlDefaultFolders)",
                              "olFolderContacts");
```

Si nous désirons appeler des méthodes dont la liste de paramètres contient des types de données non pris en charge, nous pouvons utiliser `QAxBase::queryInterface()`pour récupérer l'interface COM et appeler directement la méthode. Comme toujours avec COM, nous devons appeler `Release()` quand nous n'avons plus besoin de l'interface. S'il devient très fréquent d'avoir à appeler de telles méthodes, nous pouvons dériver `QAxObject` ou `QAxWidget` et fournir des fonctions membres qui encapsulent les appels de l'interface COM. Vous ne devez pas ignorer que les sous classes `QAxObject` et `QAxWidget` ne peuvent pas définir leurs propres propriétés, signaux, ou slots.

Nous allons maintenant détailler le module *QAxServer*. Il nous permet de transformer un programme Qt standard en serveur ActiveX. Le serveur peut être une bibliothèque partagée ou une application autonome. Les serveurs définis en tant que bibliothèques partagées sont souvent appelés serveurs in-process (in-process signifie "qui s'exécute dans l'espace de traitement d'un client") ; les applications autonomes sont nommées serveurs hors processus.

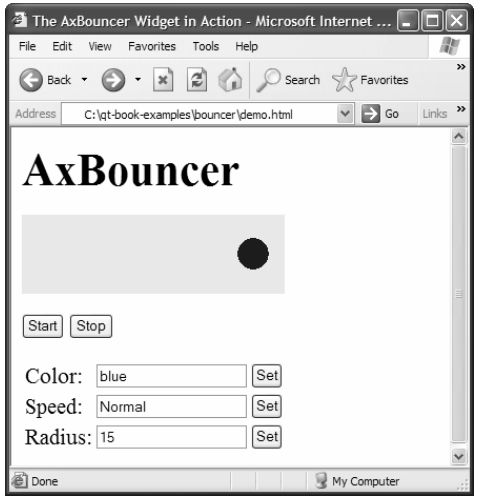
Notre premier exemple de *QAxServer* est un serveur in-process qui fournit un widget affichant une bille qui se balance de gauche à droite. Nous allons aussi expliquer comment intégrer le widget dans Internet Explorer.

Voici le début de la définition de classe du widget `AxBouncer` :

```
class AxBouncer : public QWidget, public QAxBindable
{
    Q_OBJECT
    Q_ENUMS(SpeedValue)
    Q_PROPERTY(QColor color READ color WRITE setColor)
    Q_PROPERTY(SpeedValue speed READ speed WRITE setSpeed)
    Q_PROPERTY(int radius READ radius WRITE setRadius)
    Q_PROPERTY(bool running READ isRunning)
```

`AxBouncer` hérite à la fois de `QWidget` et `QAxBindable`. La classe `QAxBindable` fournit une interface entre le widget et un client ActiveX. Tout `QWidget` peut être exporté sous forme de contrôle ActiveX, mais en dérivant `QAxBindable` nous pouvons informer le client des changements de valeur d'une propriété, et nous pouvons implémenter des interfaces COM pour compléter celles déjà implémentées par *QAxServer.*

Figure 20.6
Le widget AxBouncer dans Internet Explorer



En présence d'héritage multiple impliquant une classe dérivée de `QObject`, nous devons toujours positionner la classe dérivée de `QObject` en premier de sorte que `moc` puisque la récupérer.

Nous déclarons trois propriétés de lecture-écriture et une propriété en lecture seulement. La macro `Q_ ENUMS()` est nécessaire pour signaler à `moc` que le type `SpeedValue` est un type enum. L'énumération est déclarée dans la section publique de la classe :

```
public:
    enum SpeedValue { Slow, Normal, Fast };
```

```
    AxBouncer(QWidget *parent = 0);

    void setSpeed(SpeedValue newSpeed);
    SpeedValue speed() const { return ballSpeed; }
    void setRadius(int newRadius);
    int radius() const { return ballRadius; }
    void setColor(const QColor &newColor);
    QColor color() const { return ballColor; }
    bool isRunning() const { return myTimerId != 0; }
    QSize sizeHint() const;
    QAxAggregated *createAggregate();

public slots:
    void start();
    void stop();

signals:
    void bouncing();
```

Le constructeur de `AxBouncer` est un constructeur standard pour un widget, avec un paramètre `parent`. La macro `QAXFACTORY_DEFAULT()`, que nous utiliserons pour exporter le composant, doit recevoir un constructeur avec sa signature.

La fonction `createAggregate()` est réimplémentée dans `QAxBindable`. Nous l'étudierons un peu plus loin.

```
protected:
    void paintEvent(QPaintEvent *event);
    void timerEvent(QTimerEvent *event);

private:
    int intervalInMilliseconds() const;

    QColor ballColor;
    SpeedValue ballSpeed;
    int ballRadius;
    int myTimerId;
    int x;
    int delta;
};
```

Les sections protégées et privées de la classe sont identiques à celles que nous aurions s'il s'agissait d'un widget Qt standard.

```
AxBouncer::AxBouncer(QWidget *parent)
    : QWidget(parent)
{
    ballColor = Qt::blue;
    ballSpeed = Normal;
    ballRadius = 15;
    myTimerId = 0;
    x = 20;
    delta = 2;
}
```

Le constructeur de `AxBouncer` initialise les variables privées de la classe.

```
void AxBouncer::setColor(const QColor &newColor)
{

    if (newColor != ballColor && requestPropertyChange("color")) {
        ballColor = newColor;
        update();
        propertyChanged("color");

    }
}
```

La fonction `setColor()` définie la valeur de la propriété `color`. Elle appelle `update()` pour redessiner le widget.

Les appels de `requestPropertyChange()` et `propertyChanged()` sont plus inhabituels. Ces fonctions dérivent de `QAxBindable` et devraient normalement être appelées à chaque fois que nous modifions une propriété. `requestPropertyChange()` demande au client la permission de changer une propriété, et renvoie `true` si le client accepte. La fonction `propertyChanged()` signale au client que la propriété a changé.

Les méthodes d'accès de propriété `setSpeed()` et `setRadius()` suivent également ce modèle, ainsi que les slots `start()` et `stop()`, puisqu'ils modifient la valeur de la propriété `running`.

La fonction membre `AxBouncer` ne doit pas être oubliée :

```
QAxAggregated *AxBouncer::createAggregate()
{

      return new ObjectSafetyImpl;
}
```

La fonction `createAggregate()` est réimplémentée dans `QAxBindable`. Elle nous permet d'implémenter les interfaces COM que le module *QAxServer* n'implémente pas déjà ou de remplacer les interfaces COM par défaut de *QAxServer*. Ici, nous le faisons pour fournir l'interface `IObjectSafety`, qui est celle à partir de laquelle Internet Explorer accède aux options de sécurité du composant. Voici l'astuce standard pour se débarrasser du fameux message d'erreur "Objet non sécurisé pour le script" d'Internet Explorer.

Voici la définition de la classe qui implémente l'interface `IObjectSafety` :

```
class ObjectSafetyImpl : public QAxAggregated, public IObjectSafety
{
public:
    long queryInterface(const QUuid &iid, void **iface);

    QAXAGG_IUNKNOWN

    HRESULT WINAPI GetInterfaceSafetyOptions(REFIID riid,
            DWORD *pdwSupportedOptions, DWORD *pdwEnabledOptions);
```

```
    HRESULT WINAPI SetInterfaceSafetyOptions(REFIID riid,
            DWORD pdwSupportedOptions, DWORD pdwEnabledOptions);
};
```

La classe `ObjectSafetyImpl` hérite à la fois de `QAxAggregated` et de `IObjectSafety`. La classe `QAxAggregated` est une classe de base abstraite pour l'implémentation d'interfaces COM complémentaires. L'objet COM étendu par `QAxAggregated` est accessible par le biais de `controllingUnknown()`. Cet objet est créé en arrière-plan par le module *QAxServer*.

La macro `QAXAGG_IUNKNOWN` fournit les implémentations standard de `QueryInterface()`, `AddRef()`, et `Release()`. Ces implémentations appellent simplement la même fonction sur l'objet COM contrôleur.

```
long ObjectSafetyImpl::queryInterface(const QUuid &iid, void **iface)
{
   *iface = 0;
   if (iid == IID_IObjectSafety) {

      *iface = static_cast<IObjectSafety *>(this);
   } else {
      return E_NOINTERFACE;
   }
   AddRef();
   return S_OK;
}
```

La fonction `queryInterface()` est une fonction virtuelle pure de `QAxAggregated`. Elle est appelée par l'objet COM contrôleur afin d'offrir l'accès aux interfaces fournies par la sous classe `QAxAggregated`. Nous devons renvoyer `E_NOINTERFACE` pour les interfaces que nous n'implémentons pas et pour `IUnknown`.

```
HRESULT WINAPI ObjectSafetyImpl::GetInterfaceSafetyOptions(
        REFIID /* riid */, DWORD *pdwSupportedOptions,
        DWORD *pdwEnabledOptions)
{
    *pdwSupportedOptions = INTERFACESAFE_FOR_UNTRUSTED_DATA
                         | INTERFACESAFE_FOR_UNTRUSTED_CALLER;
    *pdwEnabledOptions = *pdwSupportedOptions;
    return S_OK;
}
HRESULT WINAPI ObjectSafetyImpl::SetInterfaceSafetyOptions(
        REFIID /* riid */, DWORD /* pdwSupportedOptions */,
        DWORD /* pdwEnabledOptions */)
{
   return S_OK;
}
```

Les fonctions `GetInterfaceSafetyOptions()` et `SetInterfaceSafetyOptions()` sont déclarées dans `IObjectSafety`. Nous les implémentons pour annoncer à tous que notre objet est bien sécurisé pour les scripts.

Examinons maintenant `main.cpp` :

```
#include <QAxFactory>

#include "axbouncer.h"

QAXFACTORY_DEFAULT(AxBouncer,
                   "{5e2461aa-a3e8-4f7a-8b04-307459a4c08c}",
                   "{533af11f-4899-43de-8b7f-2ddf588d1015}",
                   "{772c14a5-a840-4023-b79d-19549ece0cd9}",
                   "{dbce1e56-70dd-4f74-85e0-95c65d86254d}",
                   "{3f3db5e0-78ff-4e35-8a5d-3d3b96c83e09}")
```

La macro `QAXFACTORY_DEFAULT()` exporte un contrôle ActiveX. Nous pouvons l'utiliser pour les serveurs ActiveX qui exportent un seul contrôle. L'exemple suivant de cette section montre comment exporter plusieurs contrôles ActiveX.

Le premier argument destiné à `QAXFACTORY_DEFAULT()` est le nom de la classe Qt à exporter. C'est aussi le nom sous lequel le contrôle est exporté. Les cinq autres arguments sont l'ID de la classe, de l'interface, de l'interface de l'événement, de la bibliothèque des types, et de l'application. Nous pouvons générer ces identificateurs à l'aide d'un outil standard tel que `guidgen` ou `uuidgen`. Le serveur étant une bibliothèque, nous n'avons pas besoin de la fonction `main()`.

Voici le fichier `.pro` pour notre serveur ActiveX in-process :

```
TEMPLATE      = lib
CONFIG       += dll qaxserver
HEADERS       = axbouncer.h \
                objectsafetyimpl.h
SOURCES       = axbouncer.cpp \
                main.cpp \
                objectsafetyimpl.cpp
RC_FILE       = qaxserver.rc
DEF_FILE      = qaxserver.def
```

Les fichiers `qaxserver.rc` et `qaxserver.def` auxquels il est fait référence dans le fichier `.pro` sont des fichiers standards que l'on peut copier dans le répertoire `src\activeqt\control` de Qt.

Le make?le ou le fichier de projet Visual C++ généré par `qmake` contient les règles qui régissent l'enregistrement du serveur dans le registre de Windows. Pour enregistrer le serveur sur les machines utilisateur, nous pouvons faire appel à l'outil `regsvr32` disponible sur tous les systèmes Windows.

Nous incluons alors le composant Bouncer dans une page HTML *via* la balise `<object>` :

```
<object id="AxBouncer"
        classid="clsid:5e2461aa-a3e8-4f7a-8b04-307459a4c08c">
<b>The ActiveX control is not available. Make sure you have built and
registered the component server.</b>
</object>
```

Il est possible de créer des boutons qui invoquent des slots :

```
<input type="button" value="Start" onClick="AxBouncer.start()">
<input type="button" value="Stop" onClick="AxBouncer.stop()">
```

Nous pouvons manipuler le widget à l'aide de JavaScript ou VBScript comme n'importe quel autre contrôle ActiveX. Le fichier `demo.html` proposé sur le CD présente une page rudimentaire qui utilise le serveur ActiveX.

Notre dernier exemple est une application Carnet d'adresse. Elle peut se comporter comme une application standard Qt/Windows ou comme un serveur ActiveX hors processus. Cette dernière option nous permet de créer le script de l'application en Visual Basic, par exemple.

```
class AddressBook : public QMainWindow

{
    Q_OBJECT
    Q_PROPERTY(int count READ count)
    Q_CLASSINFO("ClassID", "{588141ef-110d-4beb-95ab-ee6a478b576d}")
    Q_CLASSINFO("InterfaceID", "{718780ec-b30c-4d88-83b3-79b3d9e78502}")
    Q_CLASSINFO("ToSuperClass", "AddressBook")

public:
    AddressBook(QWidget *parent = 0);
    ~AddressBook();

    int count() const;

public slots:
    ABItem *createEntry(const QString &contact);
    ABItem *findEntry(const QString &contact) const;
    ABItem *entryAt(int index) const;

private slots:
    void addEntry();
    void editEntry();
    void deleteEntry();

private:
    void createActions();
    void createMenus();

    QTreeWidget *treeWidget;
    QMenu *fileMenu;
    QMenu *editMenu;
    QAction *exitAction;
    QAction *addEntryAction;
    QAction *editEntryAction;
    QAction *deleteEntryAction;
};
```

Le widget `AddressBook` correspond à la fenêtre principale de l'application. La propriété et les slots fournis seront disponibles *via* le script. La macro `Q_CLASSINFO()` permet de spécifier la classe et les identifiants d'interface associés à cette dernière. Ceux-ci ont été générés avec un outil tel que `guid` ou `uuid`.

Dans l'exemple précédent, nous avions spécifié la classe et les identifiants d'interface quand nous avions exporté la classe `QAxBouncer` à l'aide de la macro `QAXFACTORY_DEFAULT()`. Dans cet exemple, nous allons exporter plusieurs classes, nous ne pouvons donc pas exécuter `QAXFACTORY_DEFAULT()`. Nous avons deux options :

- Dériver `QAxFactory`, réimplémenter ses fonctions virtuelles pour fournir des informations concernant les types à exporter, et exécuter la macro `QAXFACTORY_EXPORT()` pour enregistrer le composant fabricant.
- Exécuter les macros `QAXFACTORY_BEGIN()`, `QAXFACTORY_END()`, `QAXCLASS()`, et `QAXTYPE()` pour déclarer et enregistrer le composant fabricant. Cette approche exige de spécifier la classe et l'identifiant d'interface à l'aide de `Q_CLASSINFO()`.

Voici la définition de la classe `AddressBook` : La troisième occurrence de `Q_CLASSINFO()` pourrait vous sembler un peu bizarre. Par défaut, les contrôles ActiveX exposent non seulement leurs propres propriétés, signaux, et slots à leurs clients, mais aussi ceux de leurs superclasses jusqu'à `QWidget`. L'attribut `ToSuperClass` permet de spécifier la superclasse de niveau supérieur (dans l'arbre d'héritage) que nous désirons exposer. Nous spécifions ici le nom de la classe du composant (`AddressBook`) en tant que classe de niveau le plus haut à exporter, ce qui signifie que les propriétés, signaux, et slots définis dans les superclasses d'`AddressBook` ne seront pas exportés.

```
class ABItem : public QObject, public QTreeWidgetItem
{
    Q_OBJECT
    Q_PROPERTY(QString contact READ contact WRITE setContact)
    Q_PROPERTY(QString address READ address WRITE setAddress)
    Q_PROPERTY(QString phoneNumber READ phoneNumber WRITE setPhoneNumber)
    Q_CLASSINFO("ClassID", "{bc82730e-5f39-4e5c-96be-461c2cd0d282}")
    Q_CLASSINFO("InterfaceID", "{c8bc1656-870e-48a9-9937-fbe1ceff8b2e}")
    Q_CLASSINFO("ToSuperClass", "ABItem")

public:
    ABItem(QTreeWidget *treeWidget);
    void setContact(const QString &contact);
    QString contact() const { return text(0); }
    void setAddress(const QString &address);
    QString address() const { return text(1); }
    void setPhoneNumber(const QString &number);
    QString phoneNumber() const { return text(2); }

public slots:
    void remove();
};
```

La classe `ABItem` représente une entrée dans le carnet d'adresses. Elle hérite de `QTreeWidget-Item` pour pouvoir être affichée dans un `QTreeWidget` et de `QObject` pour pouvoir être exportée sous forme d'objet COM.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    if (!QAxFactory::isServer()) {
        AddressBook addressBook;
        addressBook.show();
        return app.exec();
    }
    return app.exec();
}
```

Dans `main()`, nous vérifions si l'application s'exécute en autonome ou en tant que serveur. L'option de ligne de commande `-activex` est reconnue par `QApplication` et exécute l'application en tant que serveur. Si l'application n'est pas exécutée de cette façon, nous créons le widget principal et nous l'affichons comme nous le ferions normalement pour une application Qt autonome.

En complément de `-activex`, les serveurs ActiveX comprennent les options de ligne de commande suivantes :

- `-regserver` enregistre le serveur dans le registre système.
- `-unregserver` annule l'enregistrement du serveur dans le registre système.
- `-dumpidl file` inscrit l'IDL (*Interface Definition Langage*) du serveur dans le fichier spécifié.

Lorsque l'application s'exécute en tant que serveur, nous devons exporter les classes `Address-Book` et `ABItem` en tant que composants COM :

```
QAXFACTORY_BEGIN("{2b2b6f3e-86cf-4c49-9df5-80483b47f17b}",
                 "{8e827b25-148b-4307-ba7d-23f275244818}")
QAXCLASS(AddressBook)
QAXTYPE(ABItem)
QAXFACTORY_END()
```

Les macros précédentes exportent un composant fabricant d'objets COM. Puisque nous devons exporter deux types d'objets COM, nous ne pouvons pas nous contenter d'exécuter `QAXFACTORY_DEFAULT()` comme nous l'avions fait dans l'exemple précédent.

Le premier argument de `QAXFACTORY_BEGIN()` correspond à l'ID de bibliothèque de types ; le second est l'ID de l'application. Entre `QAXFACTORY_BEGIN()` et `QAXFACTORY_ END()`, nous spécifions toutes les classes pouvant être instanciées et tous les types de données qui ont besoin d'être accessibles sous forme d'objets COM.

Voici le fichier `.pro` pour notre serveur ActiveX hors processus :

```
TEMPLATE       = app
CONFIG        += qaxserver
```

```
HEADERS       = abitem.h \
                addressbook.h \
                editdialog.h
SOURCES       = abitem.cpp \
                addressbook.cpp \
                editdialog.cpp \
                main.cpp
FORMS         = editdialog.ui
RC_FILE       = qaxserver.rc
```

Le fichier `qaxserver.rc` auquel il est fait référence dans le fichier `.pro` est un fichier standard que l'on peut copier dans le répertoire `src\activeqt\control` de Qt.

Cherchez dans le répertoire `vb` de l'exemple un projet Visual Basic qui utilise le serveur de carnet d'adresses.

Nous en avons terminé avec la présentation du framework ActiveQt. La distribution de Qt propose des exemples supplémentaires, et la documentation contient des informations concernant la façon de créer les modules *QAxContainer* et *QAxServer* et comment résoudre les problèmes d'interopérabilité courants.

Prendre en charge la gestion de session X11

Lorsque nous quittons X11, certains gestionnaires de fenêtre nous demandent si nous désirons enregistrer la session. Si nous répondons oui, les applications en cours d'exécution seront automatiquement redémarrées lors de la prochaine ouverture de session, au même emplacement sur l'écran et, cerise sur le gâteau, dans le même état.

Le composant X11 chargé de l'enregistrement et de la restauration de la session est le *gestionnaire de session*. Pour qu'une application Qt/X11 puisse être prise en charge par ce gestionnaire, nous devons réimplémenter `QApplication::saveState()` afin d'enregistrer l'état de l'application.

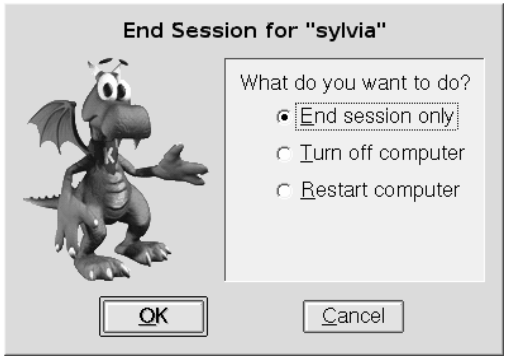


Figure 20.7
Fermeture de session sous KDE

Windows 2000 et XP, et certains systèmes Unix, proposent un autre mécanisme nommé Mise en veille. Dès que l'utilisateur active la mise en veille, le système d'exploitation sauvegarde simplement la mémoire de l'ordinateur sur disque puis la recharge au moment de la réactivation. Les applications ne sont pas sollicitées et n'ont pas besoin d'être averties de l'opération.

Lorsque l'utilisateur demande l'arrêt de l'ordinateur, nous pouvons prendre le contrôle juste avant l'exécution de cette opération en réimplémentant `QApplication::commitData()`. Nous avons ainsi la possibilité d'enregistrer toute donnée non sauvegardée et de dialoguer avec l'utilisateur si nécessaire. Cette partie de la gestion de session est prise en charge sur les deux plates-formes X11 et Windows.

Notre étude de la gestion de session va s'effectuer en analysant le code d'une application Tic-Tac-Toe compatible avec cette fonction. Commençons par examiner la fonction `main()` :

```
int main(int argc, char *argv[])
{
    Application app(argc, argv);
    TicTacToe toe;
    toe.setObjectName("toe");
    app.setTicTacToe(&toe);
    toe.show();
    return app.exec();
}
```

Nous créons un objet `Application`. La classe `Application` hérite de `QApplication` et réimplémente à la fois `commitData()` et `saveState()` afin de prendre en charge la gestion de session.

Nous créons ensuite un widget `TicTacToe`, que nous associons à l'objet `Application`, puis nous l'affichons. Nous avons appelé le widget `TicTacToe`. Nous devons attribuer des noms d'objet uniques aux widgets de niveau supérieur si nous voulons que le gestionnaire de session soit en mesure de restaurer les tailles et positions des fenêtres.

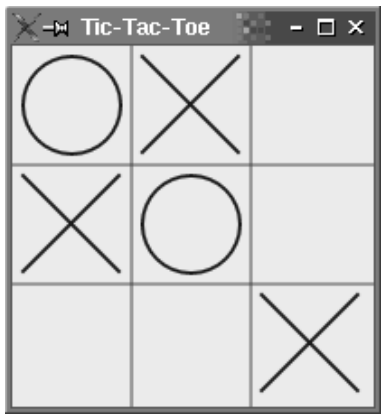


Figure 20.8
L'application Tic-Tac-Toe

Voici la définition de la classe `Application` :

```
class Application : public QApplication
{
    Q_OBJECT

public:
    Application(int &argc, char *argv[]);

    void setTicTacToe(TicTacToe *tic);
    void saveState(QSessionManager &sessionManager);
    void commitData(QSessionManager &sessionManager);

private:
    TicTacToe *ticTacToe;
};
```

La classe `Application` stocke un pointeur vers le widget `TicTacToe` dans une variable privée.

```
void Application::saveState(QSessionManager &sessionManager)
{
    QString fileName = ticTacToe->saveState();

    QStringList discardCommand;
    discardCommand << "rm" << fileName;
    sessionManager.setDiscardCommand(discardCommand);
}
```

Sous X11, la fonction `saveState()` est appelée au moment où le gestionnaire de session veux que l'application enregistre son état. La fonction est aussi disponible sur d'autres plates-formes, mais elle n'est jamais appelée. Le paramètre `QSessionManager` nous permet de communiquer avec le gestionnaire de session.

Nous commençons par demander au widget `TicTacToe` d'enregistrer son état dans un fichier. Nous affectons ensuite une valeur à la *commande d'annulation* du gestionnaire. Cette commande est celle que le gestionnaire de session doit exécuter pour supprimer toute information stockée concernant l'état courant. Pour cet exemple, nous la définissons en

```
rm sessionfile
```

où `sessionfile` est le nom du fichier qui contient l'état enregistré pour la session, et `rm` est la commande Unix standard pour supprimer des fichiers.

Le gestionnaire de session comporte aussi une *commande de redémarrage*. Il s'agit de celle que le gestionnaire exécute pour redémarrer l'application. Par défaut, Qt fournit la commande de redémarrage suivante :

```
appname -session id_key
```

La première partie, `appname`, est dérivée de `argv[0]`. Le composant `id` correspond à l'identifiant de session fourni par le gestionnaire de session; dont l'unicité est garantie au sein de plusieurs applications et de différentes exécutions de la même application. La partie `key` est ajoutée afin d'identifier de façon unique l'heure à laquelle l'état a été enregistré. Pour diverses raisons, le gestionnaire de session peut appeler plusieurs fois `saveState()` au cours d'une même session, et les différents états doivent pouvoir être distingués.

Etant donné les limites des gestionnaires de session existants, nous devons nous assurer que le répertoire de l'application se trouve dans la variable d'environnement `PATH` si nous voulons que l'application puisse redémarrer correctement. Si vous désirez en particulier tester l'exemple Tic-Tac-Toe, vous devez l'installer dans le répertoire `/usr/bin` par exemple et l'invoquer en tapant `tictactoe`.

Pour des applications simples, comme Tic-Tac-Toe, nous pourrions enregistrer l'état sous forme d'argument de ligne de commande supplémentaire de la commande de redémarrage. Par exemple :

```
tictactoe -state OX-XO-X-O
```

Ceci nous éviterait d'avoir à stocker les données dans un fichier puis à fournir une commande d'annulation pour supprimer le fichier.

```
void Application::commitData(QSessionManager &sessionManager)
{
    if (ticTacToe->gameInProgress()
            && sessionManager.allowsInteraction()) {
        int r = QMessageBox::warning(ticTacToe, tr("Tic-Tac-Toe"),
                        tr("The game hasn't finished.\n"
                           "Do you really want to quit?"),
                        QMessageBox::Yes | QMessageBox::Default,
                        QMessageBox::No | QMessageBox::Escape);
       if (r == QMessageBox::Yes) {
           sessionManager.release();
       } else {
           sessionManager.cancel();
       }
    }
}
```

La fonction `commitData()` est appelée quand l'utilisateur ferme la session. Nous pouvons la réimplémenter de sorte d'afficher un message d'avertissement qui signale à l'utilisateur le risque de perte de données. L'implémentation par défaut ferme tous les widgets de niveau le plus haut, ce qui donne le même résultat que lorsque l'utilisateur ferme les fenêtres l'une après l'autre en cliquant sur le bouton de fermeture de leur barre de titre. Au Chapitre 3, nous avons vu comment réimplémenter `closeEvent()` pour détecter cette situation et afficher un message.

Pour cet exemple, nous allons réimplémenter `commitData()` et afficher un message demandant à l'utilisateur de confirmer la fermeture de session si un jeu est en cours d'exécution et si le gestionnaire de session nous permet de dialoguer avec l'utilisateur. Si l'utilisateur clique sur

Yes, nous appelons `release()` pour ordonner au gestionnaire de poursuivre la fermeture de session ; s'il clique sur No, nous appelons `cancel()` pour annuler l'opération.

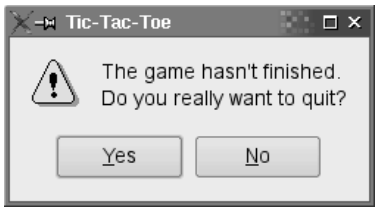


Figure 20.9
"Vous désirez vraiment quitter ?"

Examinons maintenant la classe `TicTacToe` :

```
class TicTacToe : public QWidget
{
    Q_OBJECT

public:
    TicTacToe(QWidget *parent = 0);

    bool gameInProgress() const;
    QString saveState() const;
    QSize sizeHint() const;

protected:
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);

private:
    enum { Empty = '-', Cross = 'X', Nought = 'O' };

    void clearBoard();
    void restoreState();
    QString sessionFileName() const;
    QRect cellRect(int row, int column) const;
    int cellWidth() const { return width() / 3; }
    int cellHeight() const { return height() / 3; }
    bool threeInARow(int row1, int col1, int row3, int col3) const;

    char board[3][3];
    int turnNumber;
};
```

La classe `TicTacToe` hérite de `QWidget` et réimplémente `sizeHint()`, `paintEvent()`, et `mousePressEvent()`. Elle fournit aussi les fonctions `gameInProgress()` et `saveState()` que nous avions utilisées dans notre classe `Application`.

```
TicTacToe::TicTacToe(QWidget *parent)
    : QWidget(parent)
{
```

```
    clearBoard();
    if (qApp->isSessionRestored())
        restoreState();

    setWindowTitle(tr("Tic-Tac-Toe"));
}
```

Dans le constructeur, nous effaçons le tableau et si l'application avait été invoquée avec l'option `-session`, nous appelons la fonction privée `restoreState()` pour recharger l'ancienne session.

```
void TicTacToe::clearBoard()
{
    for (int row = 0; row < 3; ++row) {
        for (int column = 0; column < 3; ++column) {
            board[row][column] = Empty;
        }
    }
    turnNumber = 0;
}
```

Dans `clearBoard()`, nous effaçons toutes les cellules et nous définissons `turnNumber` à 0.

```
QString TicTacToe::saveState() const
{
    QFile file(sessionFileName());
    if (file.open(QIODevice::WriteOnly)) {
        QTextStream out(&file);
        for (int row = 0; row < 3; ++row) {
            for (int column = 0; column < 3; ++column)
                out << board[row][column];
        }
    }
    return file.fileName();
}
```

Dans `saveState()`, nous enregistrons l'état du tableau sur disque. Le format est simple, avec 'X' pour les croix, 'O' pour les ronds, et '-+-' pour les cellules vides.

```
QString TicTacToe::sessionFileName() const
{
    return QDir::homePath() + "/.tictactoe_" + qApp->sessionId() + "_"
           + qApp->sessionKey();
}
```

La fonction privée `sessionFileName()` renvoie le nom de fichier pour l'ID et la clé de session en cours. Cette fonction est exploitée à la fois par `saveState()` et par `restoreState()`. Le nom du fichier est constitué à partir de ces ID et clé de session.

```
void TicTacToe::restoreState()
{
    QFile file(sessionFileName());
```

```
        if (file.open(QIODevice::ReadOnly)) {
            QTextStream in(&file);
            for (int row = 0; row < 3; ++row) {
                for (int column = 0; column < 3; ++column) {
                    in >> board[row][column];
                    if (board[row][column] != Empty)
                        ++turnNumber;
                }
            }
        }
        update();
    }
```

Dans `restoreState()`, nous chargeons le fichier qui correspond à la session restaurée et nous remplissons le tableau avec ces informations. Nous déduisons la valeur de `turnNumber` à partir du nombre de `X` et `O` sur le tableau.

Dans le constructeur de `TicTacToe`, nous appelons `restoreState()` lorsque `QApplication::isSessionRestored()` renvoie `true`. Dans ce cas, `sessionId()` et `sessionKey()` renvoient les mêmes valeurs que lorsque l'état de l'application était enregistré, et `sessionFileName()` renvoie le nom de fichier pour cette session.

Les tests et le débogage de la gestion de session peuvent être pénibles, parce que vous devez continuellement vous connecter puis vous déconnecter. Un moyen d'éviter ces opérations consiste à utiliser l'utilitaire standard `xsm` fourni avec X11. Le premier appel de `xsm` ouvre une fenêtre du gestionnaire de session et un terminal. Les applications démarrées dans ce terminal vont toutes utiliser `xsm` comme gestionnaire de session plutôt que celui du système. Nous pouvons alors nous servir de la fenêtre de `xsm` pour terminer, redémarrer, ou supprimer une session, et voir si notre application se comporte normalement. Vous trouverez tous les détails de cette procédure à l'adresse **http://doc.trolltech.com/4.1/session.html**.

21

Programmation embarquée

Au sommaire de ce chapitre

- ✔ Démarrer avec Qtopia
- ✔ Personnaliser Qtopia Core

Le développement des logiciels destinés à s'exécuter sur des périphériques mobiles tels que les PDA et les téléphones portables présente des difficultés bien spécifiques parce que les systèmes embarqués possèdent généralement des processeurs plus lents, une capacité de mémoire permanente réduites (mémoire flash ou disque dur), moins de mémoire et un écran plus petit que les ordinateurs de bureau.

Qtopia Core (précédemment nommé Qt/Embedded) est une version de Qt optimisée pour le système d'exploitation Linux embarqué. Qtopia Core fournit les mêmes outils et la même API que les versions de bureau de Qt (Qt/Windows, Qt/X 11 et Qt/Mac) complétés des classes et outils requis pour la programmation embarquée. Par le biais d'une double licence, ce système est disponible à la fois pour le développement open source et le développement commercial.

Qtopia Core peut s'exécuter sur n'importe quel matériel équipé de Linux (notamment les architectures Intel x86, MIPS, ARM, StrongARM, Motorola 68000, et PowerPC). Il comporte une mémoire d'image et prend en charge un compilateur C++. Contrairement à Qt/X11, il n'a pas besoin du système XWindow ; en fait, il implémente son propre système de fenêtrage (QWS) ce qui permet d'optimiser au maximum la gestion des mémoires. Pour réduire encore ses besoins en mémoire, Qtopia Core peut être recompilé en excluant les fonctions non utilisées. Si les applications et composants exploités sur un périphérique sont connus par avance, ils peuvent être compilés ensemble pour fournir un seul exécutable avec des liens statiques vers les bibliothèques de Qtopia Core.

Qtopia Core bénéficie également de diverses fonctionnalités qui existent aussi dans les versions bureau de Qt, notamment l'usage extensif du partage de données implicite ("copie lors de l'écriture") pour ce qui concerne la technique d'optimisation de la mémoire, la prise en charge des styles de widget personnalisés via QStyle et un système de disposition qui s'adapte pour optimiser l'espace écran disponible.

Qtopia Core est au cœur de l'offre embarquée de Trolltech, qui comprend également Qtopia Platform, Qtopia PDA, et Qtopia Phone. Ces versions fournissent les classes et applications conçues spécifiquement pour les périphériques portables et elles peuvent être intégrées avec plusieurs machines virtuelles Java tiers.

Démarrer avec Qtopia

Les applications de Qtopia Core peuvent être développées sur toute plate-forme équipée d'une chaîne d'outils multiplate-forme. L'option la plus courante consiste à installer un compilateur croisé GNU C++ sur un système UNIX. Ce processus et simplifié par un script et un ensemble de correctifs fournis par Dan Kegel à l'adresse **http://kegel.com/crosstool/.** Puisque Qtopia Core contient l'API de Qt, il est généralement possible de travailler avec une version bureau de Qt, telle que Qt/X11 ou Qt/Windows pour la plupart des développements.

Le système de configuration de Qtopia Core prend en charge les compilateurs croisés, via l'option `-embedded` du script `configure`. Par exemple, pour obtenir une génération destinée à l'architecture ARM, vous devriez saisir

```
./configure -embedded arm
```

Nous avons la possibilité de créer des configurations personnalisées en ajoutant de nouveaux fichiers dans le répertoire `mkspecs/ qws` de Qt.

Qtopia Core dessine directement dans la mémoire d'image de Linux (la zone de mémoire associée avec l'affichage vidéo). Pour accéder à cette mémoire d'image, vous devrez accorder des permissions en écriture au périphérique `/dev/fb0`.

Pour exécuter les applications de Qtopia Core, nous devons commencer par démarrer un processus qui joue le rôle de serveur. Celui-ci est chargé d'allouer des zones d'écran aux clients et de générer les événements de souris et de clavier. Toute application Qtopia Core peut devenir

serveur si vous spécifiez `-qws` sur sa ligne de commande ou si vous transmettez `QApplication::GuiServer` comme troisième paramètre du constructeur de `QApplication`.

Les applications client communiquent avec le serveur Qtopia Core par le biais de la mémoire partagée. En arrière plan, les clients se dessinent eux-mêmes dans cette mémoire partagée et sont chargés d'afficher leurs propres décorations de fenêtre. Cela permet d'obtenir un niveau de communication minimum entre les clients et le serveur tout en proposant une interface utilisateur soignée. Les applications de Qtopia Core s'appuient normalement sur `QPainter` pour se dessiner elles-mêmes mais elles peuvent aussi accéder au matériel vidéo directement à l'aide de `QDirectPainter`.

Les clients ont la possibilité de communiquer *via* le protocole QCOP. Un client peut écouter sur un canal nommé en créant un objet `QCopChannel` et en se connectant à son signal `received()`. Par exemple :

```
QCopChannel *channel = new QCopChannel("System", this);
connect(channel, SIGNAL(received(const QString &, const QByteArray &)),
        this, SLOT(received(const QString &, const QByteArray &)));
```

Un message QCOP est constitué d'un nom et éventuellement d'un `QByteArray`. La fonction `QCopChannel::send()` statique diffuse un message sur le canal. Par exemple :

```
QByteArray data;
QDataStream out(&data, QIODevice::WriteOnly);
out << QDateTime::currentDateTime();
QCopChannel::send("System", "clockSkew(QDateTime)", data);
```

L'exemple précédent illustre un idiome connu : nous nous servons de `QDataStream` pour coder les données, et pour garantir que le `QByteArray` sera correctement interprété par le destinataire, nous joignons le format de données dans le nom du message comme s'il s'agissait d'une fonction C++.

Plusieurs variables d'environnement affectent les applications de Qtopia Core. Les plus importantes sont `QWS_MOUSE_PROTO` et `QWS_KEYBOARD`, qui spécifient le périphérique souris et le type de clavier. Vous trouverez une liste complète des variables d'environnement sur la page **http://doc.trolltech.com/4.1/emb-envvars. html**.

Si UNIX est la plate-forme de développement, nous pouvons tester l'application en utilisant la mémoire d'image virtuelle de Qtopia (`qvfb`), une application X11 qui simule pixel par pixel la mémoire d'image réelle. Cela accélère considérablement le cycle de développement. Pour activer la prise en charge de la mémoire virtuelle dans Qtopia Core, vous transmettez l'option `-qvfb` au script `configure`. N'oubliez pas que cette option n'est pas destinée à un usage en production. L'application de mémoire d'image virtuelle se trouve dans le répertoire `tools/qvfb` et peut être invoquée de la façon suivante :

```
qvfb -width 320 -height 480 -depth 32
```

Une autre option qui fonctionne sur la plupart des plates-formes consiste à utiliser VNC (*Virtual Network Computing*) pour exécuter des applications à distance. Pour activer la prise en

charge de VNC dans Qtopia Core, vous transmettez l'option `-qt-gfx-vnc` à `configure`. Lancez ensuite vos applications Qtopia Core avec l'option de ligne de commande `-display VNC:0` et exécutez un client VNC qui pointe sur l'hôte sur lequel vos applications s'exécutent. La taille et la résolution de l'écran peuvent être spécifiés en définissant les variables d'environnement `QWS_SIZE` et `QWS_DEPTH` sur l'hôte qui exécute les applications Qtopia Core (par exemple, `QWS_SIZE=320x480` et `QWS_DEPTH=32`).

Personnaliser Qtopia Core

A l'installation de Qtopia Core, nous pouvons spécifier les fonctionnalités dont nous n'avons pas besoin afin de réduire l'occupation mémoire. Qtopia Core comprend plus d'une centaine de fonctionnalités configurables, chacune étant associée à un symbole de préprocesseur. `QT_NO_FILEDIALOG`, par exemple, exclut `QFileDialog` de la bibliothèque *QtGui*, et `QT_NO_I18N` renonce à la prise en charge de l'internationalisation. Les fonctionnalités sont énumérées dans le fichier `src/ corelib/qfeatures.txt`.

Qtopia Core propose cinq configurations type (`minimum`, `small`, `medium`, `large`, et `dist`) qui sont stockées dans les fichiers `src/corelib/qconfig_xxx.h`. Vous spécifiez ces configurations *via* l'option `-qconfig xxx` de `configure`, par exemple :

```
./configure -qconfig small
```

Pour créer des configurations personnalisées, nous pouvons fournir manuellement un fichier `qconfig-xxx.h` et l'utiliser comme s'il s'agissait d'une configuration standard. Nous pourrions aussi nous servir de l'outil graphique `qconfig`, disponible dans le sous-répertoire `tools` de Qt.

Qtopia Core propose les classes suivantes pour le dialogue avec les périphériques d'entrée et de sortie et pour personnaliser l'aspect et le comportement du système de fenêtrage :

Classe	*Classe de base pour*
`QScreen`	Pilotes d'écran
`QScreenDriverPlugin`	plug-in de pilote d'écran
`QWSMouseHandler`	Pilotes de souris
`QMouseDriverPlugin`	Plug-in de pilotes de souris
`QWSKeyboardHandler`	Pilotes de clavier
`QKbdDriverPlugin`	Plug-in de pilote de clavier
`QWSInputMethod`	Méthodes d'entrée
`QDecoration`	Styles de décoration de fenêtre
`QDecorationPlugin`	Plug-in fournissant des styles de décoration de fenêtre

Vous obtenez la liste des pilotes prédéfinis, des méthodes d'entrée, et des styles de décoration de fenêtre en exécutant le script `configure` avec l'option `-help`.

Vous spécifiez le pilote vidéo à l'aide de l'option de ligne de commande `-display` au démarrage du serveur Qtopia Core, comme expliqué dans la section précédente, ou en définissant la variable d'environnement `QWS_DISPLAY`. Vous spécifiez le pilote de souris et le périphérique associé *via* la variable d'environnement `QWS_MOUSE_PROTO`, dont la valeur suit la syntaxe *type : device*, où *type* est un des pilotes pris en charge et `device` le chemin d'accès au périphérique (par exemple, `QWS_MOUSE_PRO-TO=IntelliMouse:/dev/mouse`). Les claviers sont gérés d'une façon analogue dans la variable d'environnement `QWS_ KEYBOARD`. Les méthodes d'entrée et décorations de fenêtre sont définies par programme dans le serveur en exécutant `QWSServer::setCurrentInputMethod()` et `QApplication::qwsSetDecoration()`.

Les styles de décoration de fenêtre sont définis indépendamment du style de widget, qui hérite de `QStyle`. Il est tout à fait possible, par exemple, de définir Windows comme style de décoration de fenêtre et Plastique comme style de widget. Si vous en avez envie, les décorations peuvent être réglées fenêtre par fenêtre.

La classe `QWSServer` fournit diverses fonctions pour personnaliser le système de fenêtrage. Les applications qui s'exécutent en tant que serveurs Qtopia Core peuvent accéder à l'instance unique `QWSServer` *via* la variable globale `qwsServer`, initialisée dans le constructeur de `QApplication`.

Qtopia Core prend en charge les formats de police suivants : TrueType (TTF), PostScript Type 1, Bitmap Distribution Format (BDF), et Qt Pre-rendered Fonts (QPF).

QPF étant un format brut, il est plus rapide et généralement plus compact que des formats vectoriels tels que TTF et Type 1 si le besoin se limite à une ou deux tailles différentes. L'outil `makeqpf` permet de créer des fichiers QPF à partir de fichiers TTF ou Type 1. Une autre solution consiste à exécuter nos applications avec l'option de ligne de commande `-savefonts`.

Au moment d'écrire ces lignes, Trolltech développe une couche supplémentaire au-dessus de Qtopia Core pour rendre le développement des applications embarquées encore plus rapide et efficace. Une prochaine édition de cet ouvrage devrait contenir de plus amples informations à ce propos.

Annexes

A *Installer Qt*

B *Introduction au langage C++ pour les programmeurs Java et C#*

A

Installer Qt

Au sommaire de ce chapitre

- ✔ A propos des licences
- ✔ Installer Qt/Windows
- ✔ Installer Qt/Mac
- ✔ Installer Qt/X11

Cette annexe explique comment installer Qt sur votre système à partir du CD qui accompagne cet ouvrage. Ce CD comporte les éditions de Qt 4.1.1 pour Windows, Mac OS X, et X11 (pour Linux et la plupart des versions d'Unix). Elles intègrent toutes SQLite, une base de données du domaine public, ainsi qu'un pilote SQLite. Les éditions de Qt fournies sur le CD vous sont proposées pour des raisons pratiques. Si vous envisagez sérieusement le développement logiciel, il est préférable de télécharger la dernière version de Qt à partir de **http://www.trolltech.com/download/** ou d'acheter une version commercialisée.

Trolltech fournit aussi Qtopia Core pour la création d'applications destinées aux périphériques embarqués équipés de Linux tels que les PDA et les téléphones portables. Si vous envisagez de créer des applications embarquées, vous pouvez obtenir Qtopia Core depuis la page Web de téléchargement de Trolltech.

Les exemples d'application étudiés dans cet ouvrage se trouvent dans le répertoire examples du CD. De plus, Qt propose de nombreux petits exemples d'application dans le sous répertoire examples.

A propos des licences

Qt est proposé sous deux formes : open source et commerciale. Les éditions open source sont disponibles gratuitement, alors que vous devez payer pour les éditions commerciales.

Le logiciel du CD convient pour créer des applications destinées à votre usage personnel ou pour votre formation.

Si vous désirez distribuer les applications que vous allez créer avec la version Open source de Qt, vous devez respecter les termes et conditions de licence spécifiques au logiciel que vous utilisez pour créer ces applications. Pour les éditions Open source, ces termes et conditions impliquent l'utilisation de la licence GNU *General Public License* (GPL). Les licences libres telles que la licence GPL accordent des droits aux utilisateurs des applications, notamment celui de visualiser et de modifier le code source et de distribuer les applications (sous les mêmes termes). Si vous désirez distribuer vos applications sans le code source ou si vous voulez appliquer vos propres conditions commerciales, vous devez acheter les éditions commerciales du logiciel qui sert à créer vos applications. Ces éditions vous permettent en effet de vendre et de distribuer vos applications sous vos propres termes.

Le CD contient les versions GPL de Qt pour Windows, Mac OS X, et X11. Le texte légal complet des licences est inclus avec les packages sur le CD, ainsi que les informations concernant la façon d'obtenir les versions commerciales.

Installer Qt/Windows

Lorsque vous insérez le CD sur un ordinateur équipé de Windows, le programme d'installation devrait démarrer automatiquement. Sinon, ouvrez l'explorateur de fichiers pour localiser le répertoire racine du CD et double-cliquez sur `install.exe`. (Il est possible que ce programme se nomme `install` selon la façon dont votre système est configuré.)

Si vous disposez déjà du compilateur MinGW C++ vous devez préciser le répertoire dans lequel il est installé ; sinon cochez la case afin d'installer aussi ce programme. La version GPL de Qt fournie sur le CD ne fonctionnera pas avec Visual C++, vous devez donc absolument installer MinGW si vous ne l'avez pas encore fait. Le programme d'installation vous propose également d'installer les exemples qui accompagnent cet ouvrage. Les exemples standard de Qt sont eux automatiquement installés ainsi que la documentation.

Si vous choisissez d'installer le compilateur MinGW, vous constaterez certainement un délai entre la fin de l'installation de ce dernier et le début de l'installation de Qt.

Après l'installation, un nouveau dossier apparaîtra dans le menu Démarrer intitulé "Qt by Trolltech v4.1.1 (OpenSource)". Ce dossier propose des raccourcis vers *Qt Assistant* et *Qt Designer*, et un troisième nommé "Qt 4.1.1 Command Prompt" qui démarre une fenêtre de

console. Dès que vous ouvrez cette fenêtre, elle va définir les variables d'environnement pour la compilation des programmes Qt avec MinGW. C'est dans cette fenêtre que vous allez exécuter `qmake` et `make` afin de générer vos applications Qt.

Installer Qt/Mac

Avant d'installer Qt sur Mac OS X, vous devez d'abord avoir installé le jeu d'outils Xcode Tools d'Apple. Le CD (ou le DVD) contenant ces outils est généralement fourni avec Mac OS X ; vous pouvez aussi les télécharger à partir du site Apple Developer Connection, à l'adresse **http://developer.apple.com**.

Si vous travaillez avec Mac OS X 10.4 (Tiger) et Xcode Tools 2.x (avec GCC 4.0.x), vous pouvez exécuter le programme d'installation décrit précédemment. Si vous possédez une version plus ancienne de Mac OS X, ou une version plus ancienne de GCC, vous devrez installer manuellement le package source. Celui-ci se nomme `qt-mac-opensource-4.1.1.tar.gz` et il est stocké dans le dossier mac sur le CD. Si vous l'installez, suivez les instructions de la section suivante qui concernent l'installation de Qt sous X11.

Pour exécuter le programme d'installation, insérez le CD et double-cliquez sur le package nommé `Qt.mpkg`. Le programme d'installation, `Installer.app`, va se lancer et Qt sera installé avec les exemples standard, la documentation, et les exemples associés à cet ouvrage. Cette installation s'effectue dans le répertoire `/Developer`, et les exemples du livre sont enregistrés dans le répertoire `/Developer/Examples/Qt4Book`.

Pour exécuter des commandes telles que `qmake` et `make`, vous devrez avoir recours à une fenêtre terminal, par exemple, `Terminal.app` dans `/Applications/Utilities`. Vous avez aussi la possibilité de générer des projets Xcode à l'aide de `qmake`. Pour générer, par exemple, un projet Xcode pour l'exemple hello, démarrez une console telle que `Terminal.app`, placez-vous sur le répertoire `/Developer/Examples/Qt4Book/chap01/hello`, puis saisissez la commande suivante :

```
qmake -spec macx-xcode hello.pro
```

Installer Qt/X11

Pour installer Qt sur son emplacement par défaut sous X11, vous devez être connecté en tant que root. Si vous n'avez pas ce niveau d'accès, spécifiez l'argument `-prefix` de `configure` pour indiquer un répertoire dans lequel vous avez l'autorisation d'écrire.

1. Placez-vous sur un répertoire temporaire. Par exemple :

```
cd /tmp
```

2. Décompressez le fichier archive du CD :

```
cp /cdrom/x11/qt-x11-opensource-src-4.1.1.tgz .
gunzip qt-x11-opensource-src-4.1.1.tgz
tar xvf qt-x11-opensource-src-4.1.1.tar
```

Vous allez ainsi créer le répertoire `/tmp/qt-x11-opensource-src-4.1.1`, en supposant que votre CD-ROM soit monté en `/cdrom`. Qt exige le logiciel GNU tar; qui se nomme `gtar` sur certains systèmes.

3. Exécutez l'outil `configure` avec vos options favorites afin de générer la bibliothèque de Qt et les outils qui accompagnent ce framework :

```
cd /tmp/qt-x11-opensource-src-4.1.1
./configure
```

Vous pouvez exécuter `./configure -help` pour obtenir une liste des options de configuration.

4. Pour générer Qt, saisissez

```
make
```

Cette commande va créer la bibliothèque et compiler toutes les démos, les exemples et les outils. Sur certains systèmes, `make` se nomme `gmake`.

5. Pour installer Qt, saisissez

```
su -c "make install"
```

puis saisissez le mot de passe root. Ceci va installer Qt dans `/usr/local/Troll-tech/Qt-4.1.1`. Vous pouvez choisir un autre répertoire de destination *via* l'option `-prefix` de `configure`, et si vous disposez d'un accès en écriture sur ce répertoire il vous suffit de saisir :

```
make install
```

6. Définissez certaines variables d'environnement pour Qt.

Si vous travailler avec le shell bash, ksh, zsh, ou sh, ajoutez les lignes suivantes dans votre fichier `.profile` :

```
PATH=/usr/local/Trolltech/Qt-4.1.1/bin:$PATH
export PATH
```

Si vous travaillez avec le shell csh ou tcsh, ajoutez la ligne suivante dans votre fichier `.login` :

```
setenv PATH /usr/local/Trolltech/Qt-4.1.1/bin:$PATH
```

Si vous aviez précisé `-prefix` avec `configure`, utilisez le chemin indiqué plutôt que le chemin par défaut de la ligne précédente.

Si votre compilateur ne prend pas en charge `rpath`, vous devez aussi étendre la variable d'environnement `LD_LIBRARY_PATH` pour inclure /`usr/local/ Trolltech/Qt-4.1.1/lib`. Ce n'est pas nécessaire sous Linux avec GCC.

Qt est distribué avec une application de démonstration, `qtdemo`, qui exploite de nombreuses fonctionnalités de la bibliothèque. Elle représente un bon point de départ pour tester les possibilités de Qt. Vous pouvez consulter la documentation de Qt soit en visitant le site **http://doc.trolltech.com**, soit en exécutant Qt Assistant, l'application d'aide de Qt, que vous obtenez en tapant `assistant` dans une fenêtre console.

B

Introduction au langage C++ pour les programmeurs Java et C#

Au sommaire de ce chapitre

- ✔ Démarrer avec C++
- ✔ Principales différences de langage
- ✔ La bibliothèque C++ standard

Cette annexe fournit une courte introduction au langage C++ pour les développeurs qui connaissent déjà Java ou C#. Nous supposons que vous maîtrisez les concepts de l'orienté objet tels que l'héritage et le polymorphisme, et que vous désirez étudier le C++. Pour ne pas transformer cet ouvrage en bible de 1500 pages qui couvrirait la totalité du langage C++, cette annexe vous livre uniquement l'essentiel. Elle présente les techniques et connaissances de base requises pour comprendre les programmes présentés dans le reste de cet ouvrage, avec suffisamment d'informations pour développer des applications graphiques C++ multiplateforme à l'aide de Qt.

Au moment d'écrire ces lignes, C++ est la seule option réaliste pour écrire des applications graphiques orientées objet performantes, multiplateforme. Les détracteurs de ce langage soulignent généralement que Java et C#, qui ont abandonné la compatibilité avec le langage C, sont plus pratiques à utiliser ; en fait, Bjarne Stroustrup, l'inventeur

du C++, signale dans son ouvrage *The Design and Evolution of C++* que "dans le langage C++, il y a beaucoup moins de difficultés de langage à éviter".

Heureusement, lorsque nous programmons avec Qt, nous exploitons généralement un sous-ensemble de C++ qui est très proche du langage utopique envisagé par Stroustrup, ce qui nous laisse libre de nous concentrer sur le problème immédiat. De plus, Qt étend C++ sur plusieurs points de vue, par le biais de son mécanisme "signal et slot" innovant, de sa prise en charge d'Unicode et de son mot-clé `foreach`.

Dans la première section de cette annexe, nous allons voir comment combiner des fichiers de code source C++ afin d'obtenir un programme exécutable. Nous serons ainsi amenés à étudier les concepts de base de C++ tels que les unités de compilation, les fichiers d'en-tête, les fichiers d'objets, les bibliothèques et nous apprendrons à devenir familier avec le préprocesseur, le compilateur et l'éditeur de liens C++.

Nous examinerons ensuite les différences de langage les plus importantes entre C++, Java et C# : comment définir des classes, comment utiliser les pointeurs et références, comment surcharger les opérateurs, comment utiliser le préprocesseur et ainsi de suite. Même si la syntaxe de C++ semble analogue à celle de Java ou C# au premier abord, les concepts sous-jacents diffèrent de façon subtile. D'autre part, en tant que source d'inspiration pour Java et C#, le langage C++ comporte de nombreux points communs avec ces deux langages, notamment les types de données similaires, les mêmes opérateurs arithmétiques et les mêmes instructions de contrôle de flux de base.

La dernière section est dédiée à la bibliothèque C++ standard qui fournit une fonctionnalité prêt à l'emploi que vous pouvez exploiter dans tout programme C++. Cette bibliothèque résulte de plus de 30 années de développement, et en tant que telle fournit une large gamme d'approches comprenant les styles de programmation procédural, orienté objet, et fonctionnel, ainsi que des macros et des modèles. Comparée aux bibliothèques Java et C#, la portée de la bibliothèque C++ standard est relativement limitée ; elle ne fournit aucun support au niveau de l'interface graphique GUI, de la programmation multithread, des bases de données, de l'internationalisation, de la gestion de réseau, XML et Unicode. Pour étendre la portée de C++ dans ces domaines, les développeurs C++ doivent avoir recours à diverses bibliothèques (souvent spécifiques à la plate-forme).

C'est à ce niveau que Qt apporte son bonus. Qt a démarré comme "boîte à outils" GUI multiplateforme (un ensemble de classes qui permettent d'écrire des applications à interface utilisateur graphique portables) mais qui a rapidement évolué en framework à part entière qui étend et remplace partiellement la bibliothèque C++ standard. Bien que cet ouvrage traite de Qt, il est intéressant de connaître ce que la bibliothèque C++ standard a à offrir, puisque vous risquez d'avoir besoin de travailler avec du code qui l'utilise.

Démarrer avec C++

Un programme C++ est constitué d'une ou plusieurs *unités de compilation*. Chacune de ces unités est un fichier de code source distinct, typiquement avec une extension `.cpp` (les autres extensions courantes sont `.cc` et `.cxx`) que le compilateur traite dans le même cycle d'exécution.

Pour chaque unité de compilation, le compilateur génère un *fichier objet*, avec l'extension `.obj` (sous Windows) ou `.o` (sous Unix et Mac OS X). Le fichier objet est un fichier binaire qui contient le code machine destiné à l'architecture sur laquelle le programme va s'exécuter.

Une fois que les fichiers `.cpp` ont été compilés, nous pouvons combiner les fichiers objet fin de créer un exécutable à l'aide d'un programme particulier nommé *éditeur de liens*. Cet éditeur de liens concatène les fichiers objet et résout les adresses mémoire des fonctions et autres symboles auxquels font référence les unités de compilation.

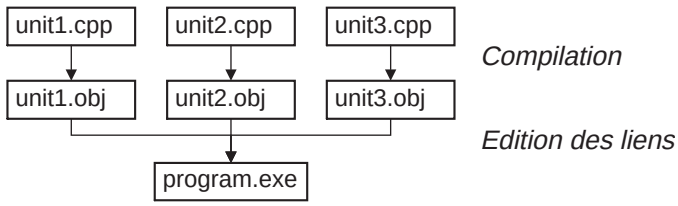


Figure B.1
Le processus de compilation de C++ (sous Windows)

Lorsque vous générez un programme, une seule unité de compilation doit contenir la fonction `main()` qui joue le rôle de point d'entrée dans le programme. Cette fonction n'appartient à aucune classe; il s'agit d'une *fonction globale*.

Contrairement à Java, pour lequel chaque fichier source doit contenir une classe exactement, C++ nous laisse libre d'organiser nos unités de compilation. Nous avons la possibilité d'implémenter plusieurs classes dans le même fichier `.cpp`, ou de répartir l'implémentation d'une classe dans plusieurs fichiers `.cpp`. Nous pouvons aussi choisir n'importe quel nom pour nos fichiers source. Quand nous effectuons une modification dans un fichier `.cpp` particulier, il suffit de recompiler uniquement ce fichier puis d'exécuter de nouveau l'éditeur de liens sur application pour créer un nouvel exécutable.

Avant de poursuivre, examinons rapidement le code source d'un programme C++ très simple qui calcule le carré d'un entier. Le programme est constitué de deux unités de compilation : `main.cpp` et `square.cpp`.

Voici `square.cpp` :

```
1  double square(double n)
2  {
3     return n * n;
4   }
```

Ce fichier contient simplement une fonction globale nommée `square()` qui renvoie le carré de son paramètre.

Voici `main.cpp` :

```
1  #include <cstdlib>
2  #include <iostream>
```

```
3  using namespace std;

4  double square(double);

5  int main(int argc, char *argv[])
6  {
7      if (argc != 2) {
8          cerr << "Usage: square <number>" << endl;
9          return 1;
10     }
11     double n = strtod(argv[1], 0);
12     cout << "The square of " << argv[1] << " is " << square(n) << endl;
13     return 0;
14 }
```

Le fichier source de `main.cpp` contient la définition de la fonction `main()`. En C++, cette fonction reçoit en paramètres un `int` et un tableau de `char*` (un tableau de chaînes de caractères). Le nom du programme est disponible en tant que `argv[0]` et les arguments de ligne de commande en tant que `argv[1]`, `argv[2]`, …, `argv[argc -1]`. Les noms de paramètre `argc` ("nombre d'arguments") et `argv` ("valeurs des arguments") sont conventionnels. Si le programme n'a pas accès aux arguments de ligne de commande, nous pouvons définir `main()` sans paramètre.

La fonction `main()` utilise `strtod()` ("string to double"), `cout` (flux de sortie standard de C++), et `cerr` (flux d'erreur standard du C++) de la bibliothèque Standard C++ pour convertir l'argument de ligne de commande en double puis pour afficher le texte sur la console. Les chaînes, les nombres, et les marqueurs de fin de ligne (`endl`) sont transmis en sortie à l'aide de l'opérateur `<<`, qui est aussi utilisé pour les opérations avec décalage des bits. Pour obtenir cette fonctionnalité standard, il faut coder les deux directives `#include` des lignes 1 et 2.

La directive `using namespace` de la ligne 3 indique au compilateur que nous désirons importer tous les identificateurs déclarés dans l'espace de noms `std` da528ns l'espace de noms global. Cela nous permet d'écrire `strtod()`, `cout`, `cerr`, et `endl` plutôt que les versions complètement qualifiées `std::strtod()`, `std::cout`, `std::cerr`, et `std::endl`. En C++, l'opérateur `::` sépare les composants d'un nom complexe.

La déclaration de la ligne 4 est un *prototype de fonction*. Elle signale au compilateur qu'une fonction existe avec les paramètres et la valeur de retour indiqués. La fonction réelle peut se trouver dans la même unité de compilation ou dans une autre. En l'absence de ce prototype, le compilateur aurait refusé l'appel de la ligne 12. Les noms de paramètre dans les prototypes de fonction sont optionnels.

La procédure de compilation du programme varie d'une plate-forme à l'autre. Pour compiler sur Solaris, par exemple, avec le compilateur C++ de Sun, il faudrait taper les commandes suivantes :

```
CC -c main.cpp
CC -c square.cpp
ld main.o square.o -o square
```

Les deux premières lignes invoquent le compilateur afin de générer les fichiers `.o` pour les fichiers `.cpp`. La troisième ligne invoque l'éditeur de liens et génère un exécutable nommé `square`, que nous pouvons ensuite appeler de la façon suivante :

```
./square 64
```

Le programme affiche le message suivant sur la fenêtre de console :

```
The square of 64 is 4096
```

Pour compiler le programme, vous préférez peut -être obtenir l'aide de votre gourou C++ local. Si vous n'y parvenez pas, lisez quand même la suite de cette annexe sans rien compiler et suivez les instructions du Chapitre 1 pour compiler votre première application C++/Qt. Qt fournit des outils à partir desquels il n'est pas difficile de générer des applications pour n'importe quelle plate-forme.

Revenons à notre programme. Dans une application du monde réel, nous devrions normalement placer le prototype de la fonction `square()` dans un fichier séparé puis inclure ce fichier dans toutes les unités de compilation dans lesquelles nous avons besoin d'appeler la fonction. Un tel fichier est nommé *fichier d'en-tête* et il comporte généralement l'extension `.h` (`.hh`, `.hpp`, et `.hxx` sont également courantes). Si nous reprenons notre exemple avec la méthode du fichier d'en-tête, il faudrait créer le fichier `square.h` avec le contenu suivant :

```
1   #ifndef SQUARE_H
2   #define SQUARE_H
3   double square(double);
4   #endif
```

Le fichier d'en-tête réunit trois directives de préprocesseur (`#ifndef`, `#define`, et `#endif`). Elles garantissent que le fichier sera traité une seule fois, même si le fichier d'en-tête est inclus plusieurs fois dans la même unité de compilation (une situation que l'on retrouve lorsque des fichiers d'en-tête incluent d'autres fichiers d'en-tête). Par convention, le symbole de préprocesseur utilisé pour obtenir ce résultat est dérivé du nom de fichier (dans notre exemple, `SQUARE_H`). Nous reviendrons au préprocesseur un peu plus loin dans cette annexe.

Voici la syntaxe du nouveau fichier `main.cpp` :

```
 1  #include <cstdlib>
 2  #include <iostream>

 3  #include "square.h"

 4  using namespace std;

 5  int main(int argc, char *argv[])
 6  {
 7     if (argc != 2) {
 8        cerr << "Usage: square <number>" << endl;
 9        return 1;
10   }
```

```
11   double n = strtod(argv[1], 0);
12   cout << "The square of " << argv[1] << " is " << square(n) << endl;
13   return 0;
14 }
```

La directive `#include` en ligne 3 insère à cet endroit le contenu du fichier `square.h`. Les directives qui débutent par un # sont prises en charge par le préprocesseur C++ avant même le début de la compilation. Dans le temps, le préprocesseur était un programme séparé que le programmeur invoquait manuellement avant d'exécuter le compilateur. Les compilateurs modernes gèrent désormais implicitement l'étape du préprocesseur.

Les directives `#include` en lignes 1 et 2 insèrent le contenu des fichiers d'en-tête `cstdlib` et `iostream`, qui font partie de la bibliothèque C++ standard. Les fichiers d'en-tête standard n'ont pas de suffixe `.h`. Les crochets (< >) autour des noms de fichier indiquent que les fichiers d'en-tête se situent à un emplacement standard sur le système, alors que les guillemets doubles indiquent au compilateur qu'il doit examiner le répertoire courant. Les inclusions sont normalement insérées au début du fichier `.cpp`.

Contrairement aux fichiers `.cpp`, les fichiers d'en-tête ne sont pas des unités de compilation à proprement parler et ne produisent pas le moindre fichier objet. Ils ne peuvent contenir que des déclarations qui permettent à diverses unités de compilation de communiquer entre elles. Il ne serait donc pas approprié d'introduire l'implémentation de la fonction `square()` dans un tel fichier. Si nous l'avions fait dans cet exemple, cela n'aurait déclenché aucune erreur, parce que nous incluons `square.h` une seule fois, mais si nous avions inclus `square.h` dans plusieurs fichiers `.cpp`, nous aurions obtenu de multiples implémentations de `square()` (une par fichier `.cpp` dans lequel elle est incluse). L'éditeur signalerait alors la présence de définitions multiples (identiques) de `square()` et refuserait de générer l'exécutable. Inversement, si nous déclarons une fonction mais que nous oublions de l'implémenter, l'éditeur de liens signale un "symbole non résolu".

Jusqu'à présent, nous avons supposé qu'un exécutable était uniquement constitué de fichiers objet. En pratique, ils comportent aussi souvent des liaisons vers des bibliothèques qui implémentent une fonctionnalité prête à l'emploi. Il existe deux principaux types de bibliothèque :

- Les *bibliothèques statiques* qui sont directement insérées dans l'exécutable, comme s'il s'agissait de fichiers objet. Cela annule les risques de perte de la bibliothèque mais cela augmente la taille de l'exécutable.
- *Les bibliothèques dynamiques* (également nommées bibliothèques partagées ou DLL) qui sont stockées sur un emplacement standard sur l'ordinateur de l'utilisateur et qui sont automatiquement chargées au démarrage de l'application.

Pour le programme `square`, nous établissons une liaison avec la bibliothèque C++ standard, qui est implémentée en tant que bibliothèque dynamique sur la plupart des plates-formes. Qt lui-même est une collection de bibliothèques qui peuvent être générées en tant que bibliothèques statiques ou dynamiques (dynamique est l'option par défaut).

Principales différences de langage

Nous allons maintenant examiner de près les domaines dans lesquels C++ diffère de Java et C#. De nombreuses différences de langage sont dues à la nature compilée du C++ et à ses obligations en termes de performances. Ainsi, le C++ ne contrôle pas les limites de tableau au moment de l'exécution, et aucun programme de récupération de la mémoire ne réaffecte la mémoire allouée dynamiquement après usage.

Pour rester concis, nous n'aborderons pas dans cette annexe les constructions C++ qui sont pratiquement identiques à leurs équivalents Java et C#. Certains sujets C++ ne sont pas non plus couverts ici parce qu'ils ne sont pas indispensables pour la programmation avec Qt. Nous ne traitons pas, en particulier, la définition des fonctions et classes modèles, celle des types union, et les exceptions. Vous trouverez une description détaillée de l'ensemble des fonctionnalités du langage dans un ouvrage tel que *Le langage C++* de Bjarne Stroustrup ou *C++ pour les programmeurs Java* de Mark Allen Weiss.

Les types de données primitifs

Les types de données primitifs du C++ sont analogues à ceux de Java ou C#. La Figure B.2 répertorie ces types avec leur définition sur les plates-formes prises en charge par Qt 4.

Type C++	*Description*
`bool`	Valeur booléenne
`char`	Entier 8 bits
`short`	Entier 16 bits
`int`	Entier 32 bits
`long`	Entier 32 bits ou 64 bits
`long long_`	Entier 64 bits
`float`	Valeur virgule flottante 32 bits (IEEE 754)
`double`	Valeur virgule flottante 64 bits (IEEE 754)

Figure B.2
Les types primitifs de C++

Par défaut, les types `short`, `int`, `long`, et `long long` sont signés, ce qui signifie qu'ils peuvent stocker aussi bien des valeurs négatives que des valeurs positives. Si nous n'avons besoin de stocker que des nombres positifs, nous pouvons coder le mot-clé `unsigned` devant le type. Alors qu'un `short` est capable de stocker n'importe quelle valeur entre –32 768 et

+32 767, un `unsigned short` stocke une valeur entre 0 et 65 535. L'opérateur de décalage à droite >> ("remplit avec des 0") a une sémantique non signée si un des opérandes est non signé.Le type bool peut avoir les valeurs `true` et `false`. De plus, les types numériques peuvent être utilisés partout où un `bool` est attendu, `0` étant interprété comme `false` et toute valeur non nulle par `true`.

Le type `char` sert à la fois pour le stockage des caractères ASCII et pour celui des entiers 8 bits (octets). Lorsqu'il est utilisé pour un entier, celui-ci peut être signé ou non signé, selon la plate-forme. Les types `signed char` et `unsigned char` sont des alternatives non ambigües au `char`. Qt fournit un type `QChar` qui stocke des caractères Unicode 16 bits.

Les instances des types intégrés ne sont pas initialisées par défaut. Lorsque nous créons une variable `int`, sa valeur pourrait tout aussi bien être de 0 que de –209 486 515. Heureusement, les compilateurs nous avertissent pour la plupart lorsque nous tentons de lire le contenu d'une variable non initialisée. Nous pouvons faire appel à des outils tels que Rational PurifyPlus et Valgrind pour détecter les accès à de la mémoire non initialisée et les autres problèmes liés à la mémoire lors de l'exécution.

En mémoire, les types numériques (à l'exception des `long`) ont des tailles identiques sur les différentes plates-formes prises en charge par Qt, mais leur représentation varie selon l'architecture du système. Sur les architectures big-endian (telles que PowerPC et SPARC), la valeur 32 bits `0x12345678` est stockée sous la forme des quatre octets `0x12 0x34 0x56 0x78`, alors que sur les architectures little-endian (telles que Intel x86), la séquence d'octets est inversée. La différence apparaît dans les programmes qui copient des zones de mémoire sur disque ou qui envoient des données binaires sur le réseau. La classe `QDataStream` de Qt, présentée au Chapitre 12 (Entrées/Sorties), peut être utilisée pour stocker des données binaires en restant indépendant de la plate-forme.

Chez Microsoft, le type `long long` s'appelle `__int64`. Dans les programmes Qt, `qlonglong` est proposé comme une alternative compatible avec toutes les plates-formes de Qt.

Définitions de classe

Les définitions de classe en C++ sont analogues à celles de Java et C#, mais vous devez connaître plusieurs différences. Nous allons les étudier avec une série d'exemples. Commençons par une classe qui représente une paire de coordonnées (*x*, *y*) :

```
#ifndef POINT2D_H
#define POINT2D_H

class Point2D
{
Public
    Point2D() {
       xVal = 0;
       yVal = 0;
    }
```

```
    Point2D(double x, double y) {
       xVal = x;
       yVal = y;
    }

    void setX(double x) { xVal = x; }
    void setY(double y) { yVal = y; }
    double x() const { return xVal; }
    double y() const { return yVal; }

Private
    double xVal;
    double yVal;
};

#endif
```

La définition de classe précédente devrait apparaître dans un fichier d'en-tête, typiquement nommé `point2d.h`. Cet exemple illustre les caractéristiques C++ suivantes :

- Une définition de classe est divisée en sections publique, protégée et privée, et se termine par un point-virgule. Si aucune section n'est spécifiée, la section par défaut est privée. (Pour des raisons de compatibilité avec le C, C++ fournit le mot-clé `struct` identique à `class` sauf que la section par défaut est publique si aucune section n'est spécifiée.)
- La classe comporte deux constructeurs (le premier sans paramètre et le second avec deux paramètres). Si nous n'avions pas déclaré de constructeur, C++ en aurait automatiquement fourni un sans paramètre avec un corps vide.
- Les fonctions d'accès `x()` et `y()` sont déclarées comme `const`. Cela signifie qu'elles ne peuvent ni modifier les variables membres ni appeler des fonctions membres non-const (telles que `setX()` et `setY()`).

Les fonctions ci-dessus ont été implémentées inline, dans la définition de classe. Vous avez aussi la possibilité de ne fournir que les prototypes de fonction dans le fichier d'en-tête et d'implémenter les fonctions dans un fichier `.cpp`. Dans ce cas, le fichier d'en-tête aurait la syntaxe suivante :

```
#ifndef POINT2D_H
#define POINT2D_H

class Point2D
{
Public
    Point2D();
    Point2D(double x, double y);

    void setX(double x);
    void setY(double y);
    double x() const;
    double y() const;
```

```
Private
    double xVal;
    double yVal;
};

#endif
```

Les fonctions seraient alors implémentées dans `point2d.cpp` :

```
#include "point2d.h"

Point2D::Point2D()
{
    xVal = 0.0;
    yVal = 0.0;
}

Point2D::Point2D(double x, double y)
{
    xVal = x;
    yVal = y;
}

void Point2D::setX(double x)
{
    xVal = x;
}

void Point2D::setY(double y)
{
    yVal = y;
}

double Point2D::x() const
{
    return xVal;
}

double Point2D::y() const
{
    return yVal;
}
```

Nous commençons par inclure `point2d.h` parce que le compilateur a besoin de la définition de classe pour être en mesure d'analyser les implémentations de fonction membre. Nous implémentons ensuite les fonctions, en préfixant leurs noms de celui de la classe suivi de l'opérateur `::`.

Nous avons déjà expliqué comment implémenter une fonction inline et maintenant comment l'implémenter dans un fichier `.cpp`. Les deux méthodes sont équivalentes d'un point de vue sémantique, mais lorsque nous appelons une fonction qui est déclarée inline, la plupart des

compilateurs se contentent d'insérer le corps de la fonction plutôt que de générer un véritable appel de fonction. Vous obtenez généralement un code plus performant, mais la taille de l'application est supérieure. C'est pourquoi seules les fonctions très courtes devraient être implémentées de cette façon ; les autres devront plutôt être implémentées dans un fichier `.cpp`. De plus, si nous oublions d'implémenter une fonction et que nous tentons de l'appeler, l'éditeur de liens va signaler un symbole non résolu.

Essayons maintenant d'utiliser la classe.

```
#include "point2d.h"

int main()
{
    Point2D alpha;
    Point2D beta(0.666, 0.875);

    alpha.setX(beta.y());
    beta.setY(alpha.x());

    return 0;
}
```

En C++, les variables de tout type peuvent être déclarées directement sans coder `new`. La première variable est initialisée à l'aide du constructeur par défaut `Point2D` (celui qui ne reçoit aucun paramètre). La seconde variable est initialisée à l'aide du second constructeur. L'accès au membre de l'objet s'effectue *via* l'opérateur "." (point).

Les variables déclarées de cette façon se comportent comme les types primitifs de Java/C# tels que `int` et `double`. Lorsque nous codons l'opérateur d'affectation, par exemple, c'est le contenu de la variable qui est copié et non une simple référence à l'objet. Si nous modifions une variable par la suite, cette modification ne sera pas répercutée dans les variables à qui on a pu affecter la valeur de la première.

En tant que langage orienté objet, C++ prend en charge l'héritage et le polymorphisme. Nous allons étudier ces deux propriétés avec l'exemple de classe de base abstraite `Shape` et une sous-classe nommée `Circle`. Commençons par la classe de base :

```
#ifndef SHAPE_H
#define SHAPE_H

#include "point2d.h"

class Shape
{
Public
    Shape(Point2D center) { myCenter = center; }
    virtual void draw() = 0;
```

```
Protected
    Point2D myCenter;
};

#endif
```

La définition se trouve dans un fichier d'en-tête nommé `shape.h`. Cette définition faisant référence à la classe `Point2D`, nous incluons `point2d.h`.

La classe `Shape` ne possède pas de classe de base. Contrairement à Java et C#, C++ ne fournit pas de classe `Object` générique à partir de laquelle toutes les classes héritent. Qt fournit `QObject` comme classe de base naturelle pour toutes sortes d'objets.

La déclaration de la fonction `draw()` présente deux caractéristiques intéressantes : elle comporte le mot-clé `virtual`, et elle se termine par = 0. Ce mot-clé signale que la fonction pourrait être réimplémentée dans les sous-classes. Comme en C#, les fonctions membres C++ ne sont pas réimplémentables par défaut. La syntaxe bizarre =0 indique que la fonction est une *fonction virtuelle pure*, c'est-à-dire une fonction qui ne possède aucune implémentation par défaut et qui doit obligatoirement être implémentée dans les sous-classes. Le concept "d'interface" en Java et C# correspond à une classe constituée uniquement de fonctions virtuelles pures en C++.

Voici la définition de la sous-classe Circle :

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include "shape.h"

class Circle : public Shape
{
Public
    Circle(Point2D center, double radius = 0.5)
        : Shape(center) {
        myRadius = radius;
    }

    void draw() {
        //exécuter ici une action
    }

Private
    double myRadius;
};

#endif
```

La classe `Circle` hérite publiquement de `Shape`, ce qui signifie que tous les membres publics de cette dernière restent publics dans `Circle`. C++ prend aussi en charge l'héritage privé et protégé, qui limite l'accès aux membres publics et protégés de la classe de base.

Le constructeur reçoit deux paramètres : le second est facultatif et il prend la valeur 0.5 lorsqu'il n'est pas spécifié. Le constructeur transmet le paramètre correspondant au centre au

constructeur de la classe de base en appliquant une syntaxe particulière entre la signature et le corps de la fonction. Dans le corps, nous initialisons la variable membre `myRadius`. Nous aurions pu aussi grouper l'initialisation de la variable avec celle du constructeur de la classe de base sur la même ligne :

```
Circle(Point2D center, double radius = 0.5)
       : Shape(center), myRadius(radius) { }
```

D'autre part, C++ n'autorise pas l'initialisation d'une variable membre dans la définition de classe, le code suivant est donc incorrect :

```
// ne compilera pas
Private
   double myRadius = 0.5;
};
```

La fonction `draw()` possède la même signature que la fonction virtuelle `draw()` déclarée dans `Shape`. Il s'agit d'une réimplémentation et elle sera invoquée de façon polymorphique au moment où `draw()` sera appelée pour une instance de `Circle` par le biais d'un pointeur ou d'une référence de `Shape`. C++ ne fournit pas de mot-clé pour une redéfinition de fonction comme en C#. Ce langage ne comporte pas non plus de mot-clé `super` ou `base` qui fasse référence à la classe de base. Si nous avons besoin d'appeler l'implémentation de base d'une fonction, nous pouvons préfixer son nom avec celui de la classe de base suivi de l'opérateur `::`. Par exemple :

```
class LabeledCircle : public Circle
{
Public
    void draw() {
       Circle::draw();
       drawLabel();
    }
    ...
};
```

C++ prend en charge l'héritage multiple, ce qui signifie qu'une classe peut être dérivée de plusieurs classes à la fois. Voici la syntaxe :

```
class DerivedClass : public BaseClass1, public BaseClass2, ...,
                     public BaseClassN
{
    ...
};
```

Par défaut, fonctions et variables déclarées dans une classe sont associées avec les instances de cette classe. Nous avons aussi la possibilité de déclarer des fonctions membres et des variables membres statiques, que vous utilisez ensuite sans instance.

Par exemple :

```
#ifndef TRUCK_H
#define TRUCK_H

class Truck
{
Public
    Truck() { ++counter; }
    ~Truck() { --counter; }

    static int instanceCount() { return counter; }

Private
    static int counter;
};

#endif
```

La variable membre statique `counter` assure le suivi du nombre d'instances de `Truck` à un instant donné. C'est le constructeur de `Truck` qui l'incrémente. Le destructeur, que vous reconnaissez au préfixe ~, décrémente cette valeur. En C++, le destructeur est automatiquement invoqué lorsqu'une variable allouée de façon statique sort de la portée ou lorsqu'une variable allouée avec `new` est supprimée. Ce comportement est analogue à celui de la méthode `finalize()` en Java, sauf que nous pouvons l'obtenir en y faisant appel à un instant spécifique.

Une variable membre statique existe uniquement dans sa classe : il s'agit de "variables de classe" plutôt que de "variables d'instance". Chaque variable membre statique doit être définie dans un fichier `.cpp` (mais sans répéter le mot-clé `static`). Par exemple :

```
#include "truck.h"

int Truck::counter = 0;
```

Si vous ne suivez pas cette règle, vous obtiendrez une erreur de "symbole non résolu" au moment de l'édition des liens. La fonction statique `instanceCount()` est accessible depuis l'extérieur de la classe, en la préfixant avec le nom de cette dernière. Par exemple :

```
#include <iostream>

#include "truck.h"

using namespace std;

int main()
{
    Truck truck1;
    Truck truck2;

    cout << Truck::instanceCount() << " equals 2" << endl;

    return 0;
}
```

Les pointeurs

Un *pointeur* en C++ est une variable qui stocke l'adresse mémoire d'un objet (plutôt que l'objet lui-même). Java et C# ont un concept analogue, la "référence," mais avec une syntaxe différente. Nous allons commencer par étudier un exemple (peu réaliste) pour observer les pointeurs en action :

```
 1 #include "point2d.h"

 2 int main()
 3 {
 4   Point2D alpha;
 5   Point2D beta;

 6   Point2D *ptr;

 7   ptr = &alpha;
 8   ptr->setX(1.0);
 9   ptr->setY(2.5);

10   ptr = &beta;
11  ptr->setX(4.0);
12  ptr->setY(4.5);

13  ptr = 0;

14  return 0;

15 }
```

Cet exemple s'appuie sur la classe `Point2D` de la sous-section précédente. Les lignes 4 et 5 définissent deux objets de type `Point2D`. Ces objets sont initialisés à (0, 0) par le constructeur de `Point2D` par défaut.

La ligne 6 définit un pointeur vers un objet `Point2D`. La syntaxe des pointeurs place un astérisque devant le nom de la variable. Ce pointeur n'ayant pas été initialisé, il contient une adresse mémoire aléatoire. Ce problème est réglé ligne 7 par l'affectation de l'adresse d'`alpha` à ce pointeur. L'opérateur unaire `&` renvoie l'adresse mémoire d'un objet. Une adresse est typiquement une valeur entière sur 32 ou 64 bits qui spécifie le décalage d'un objet en mémoire.

En lignes 8 et 9, nous accédons à l'objet `alpha` *via* le pointeur `ptr`. `ptr` étant un pointeur et non un objet, nous devons coder l'opérateur `->` (flèche) plutôt que l'opérateur . (point).

Ligne 10, nous affectons l'adresse de `beta` au pointeur. A partir de là, toute opération sur le pointeur va affecter l'objet `beta`.

Ligne 13, le pointeur est défini en pointeur nul. C++ ne fournit pas de mot-clé pour représenter un pointeur qui ne pointe pas vers un objet; c'est pourquoi nous affectons la valeur 0 (ou la constante symbolique NULL, qui représente 0). L'emploi d'un pointeur nul entraîne aussitôt une panne accompagnée d'un message d'erreur du type "Erreur de segmentation", "Erreur de

protection générale", ou "Erreur de bus". Avec l'aide d'un débogueur, nous pouvons retrouver la ligne de code à l'origine de l'erreur.

A la fin de la fonction, l'objet `alpha` contient la paire de coordonnées (1.0, 2.5), alors que `beta` contient (4.0, 4.5).

Les pointeurs sont souvent employés pour stocker des objets alloués dynamiquement à l'aide de `new`. En jargon C++, nous disons que ces objets sont alloués sur le "tas", alors que les variables locales (les variables définies dans une fonction) sont stockées sur la "pile".

Voici un extrait de code qui illustre l'allocation de mémoire dynamique à l'aide de `new` :

```
#include "point2d.h"
int main()
{
    Point2D *point = new Point2D;
    point->setX(1.0);
    point->setY(2.5);
    delete point;

    return 0;
}
```

L'opérateur `new` renvoie l'adresse mémoire de l'objet qui vient d'être alloué. Nous stockons l'adresse dans une variable pointeur et l'accès à l'objet s'effectue *via* le pointeur. Quand nous en avons terminé avec l'objet, nous libérons la mémoire associée à l'aide de l'opérateur `delete`. Contrairement à Java et C#, C++ ne possède pas de programme de libération de la mémoire (garbage collector) ; les objets alloués dynamiquement doivent être explicitement libérés à l'aide de `delete` quand nous n'en n'avons plus besoin. Le Chapitre 2 décrit le mécanisme parent–enfant de Qt, qui simplifie largement la gestion de mémoire dans les programmes C++.

Si nous oublions d'appeler `delete`, la mémoire reste occupée jusqu'à ce que le programme se termine. Cela ne poserait aucun problème dans l'exemple précédent, parce que nous n'allouons qu'un seul objet, mais dans le cas d'un programme qui allouerait constamment de nouveaux objets, les allocations de mémoire associées se produiraient jusqu'à saturation de la mémoire de l'ordinateur. Une fois qu'un objet est supprimé, la variable pointeur contient toujours l'adresse de cet objet. Ce pointeur devient un "pointeur dans le vide" et ne doit plus être utilisé. Qt fournit un pointeur "intelligent," `QPointer<T>`, qui se définit automatiquement à 0 si le `QObject` sur lequel il pointe est supprimé.

Dans l'exemple ci-dessus, nous avons invoqué le constructeur par défaut et nous avons appelé `setX()` et `setY()` pour initialiser l'objet. Nous aurions pu faire plutôt appel au constructeur à deux paramètres :q

```
Point2D *point = new Point2D(1.0, 2.5);
```

L'exemple n'exigeait pas l'emploi de `new` et de `delete`. Nous aurions très bien pu allouer l'objet sur la pile de la façon suivante :

```
Point2D point;
point.setX(1.0);
point.setY(2.5);
```

La mémoire occupée par des objets alloués de cette façon est automatiquement récupérée à la fin du bloc dans lequel ils apparaissent.

Si nous n'avons pas l'intention de modifier l'objet par l'intermédiaire du pointeur, nous pouvons déclarer le pointeur `const`. Par exemple :

```
const Point2D *ptr = new Point2D(1.0, 2.5);
double x = ptr->x();
double y = ptr->y();

// la compilation va échouer
ptr->setX(4.0);
*ptr = Point2D(4.0, 4.5);
```

Le pointeur `ptr const` sert uniquement à appeler des fonctions membres `const` telles que `x()` et `y()`. Prenez l'habitude de déclarer vos pointeurs comme `const` quand vous ne prévoyez pas de modifier l'objet par leur intermédiaire. De plus, si l'objet lui-même est `const`, nous n'avons de toute façon pas d'autre choix que d'utiliser un pointeur `const` pour stocker son adresse. Le mot-clé `const` apporte des informations au compilateur qui est ainsi en mesure de détecter très tôt des erreurs et d'améliorer les performances. C# comporte un mot-clé `const` très similaire à celui de C++. L'équivalent Java le plus proche est `final`, mais il protège les variables uniquement contre les affectations, pas contre les appels de fonctions membres "non-`const`" sur ces dernières.

Les pointeurs fonctionnent avec les types intégrés ainsi qu'avec les classes. Dans une expression, l'opérateur unaire `*` renvoie la valeur de l'objet associé au pointeur. Par exemple :

```
int i = 10;
int j = 20;

int *p = &i;
int *q = &j;

cout << *p << " equals 10" << endl;
cout << *q << " equals 20" << endl;

*p = 40;

cout << i << " equals 40" << endl;

p = q;
*p = 100;

cout << i << " equals 40" << endl;
cout << j << " equals 100" << endl;
```

L'opérateur `->`, à partir duquel vous avez accès aux membres d'un objet *via* un pointeur, a une syntaxe très particulière. Plutôt que de coder `ptr->membre`, nous pouvons aussi écrire `(*ptr).membre`. Les parenthèses sont nécessaires parce que l'opérateur "`.`" (point) est prioritaire sur l'opérateur unaire `*`.

Les pointeurs avaient mauvaise réputation en C et C++, à tel point que la promotion de Java insiste souvent sur l'absence de pointeur dans ce langage. En réalité, les pointeurs C++ sont analogues d'un point de vue conceptuel aux références de Java et C# sauf que nous pouvons nous servir de pointeurs pour parcourir la mémoire, comme vous le découvrirez un peu plus loin dans cette section. De plus, l'inclusion des classes conteneur en "copie à l'écriture" dans Qt, ainsi que la possibilité de C++ d'instancier une classe sur la pile, signifie que nous pouvons souvent éviter d'avoir recours aux pointeurs.

Les références

En complément des pointeurs, C++ prend aussi en charge le concept de "référence".Comme un pointeur, une référence C++ stocke l'adresse d'un objet. Voici les principales différences :

- Les références sont déclarées à l'aide de `&` plutôt que `*`.
- Les références doivent être initialisées et ne peuvent être réaffectées par la suite.
- L'objet associé à une référence est directement accessible ; il n'y a pas de syntaxe particulière telle que `*` ou `->`.
- Une référence ne peut être nulle.

Les références sont surtout utilisées pour déclarer des paramètres. Par défaut, C++ applique le mécanisme de transmission par valeur lors de la transmission des paramètres, ce qui signifie que lorsqu'un argument est transmis à une fonction, celle-ci reçoit une copie entièrement nouvelle de l'objet. Voici la définition d'une fonction qui reçoit ses paramètres par le biais d'un appel par valeur :

```
#include <cstdlib>

using namespace std;

double manhattanDistance(Point2D a, Point2D b)
{
    return abs(b.x() - a.x()) + abs(b.y() - a.y());
}
```

Nous pourrions alors invoquer la fonction comme suit :

```
Point2D broadway(12.5, 40.0);
Point2D harlem(77.5, 50.0);
double distance = manhattanDistance(broadway, harlem);
```

Les programmeurs C convertis évitent les opérations de copie inutiles en déclarant leurs paramètres sous forme de pointeurs plutôt que sous forme de valeurs :

```
double manhattanDistance(const Point2D *ap, const Point2D *bp)
{
    return abs(bp->x() - ap->x()) + abs(bp->y() - ap->y());
}
```

Ils doivent ensuite transmettre les adresses plutôt que les valeurs lors de l'appel de la fonction :

```
Point2D broadway(12.5, 40.0);
Point2D harlem(77.5, 50.0);
double distance = manhattanDistance(&broadway, &harlem);
```

C++ a introduit les références afin de simplifier la syntaxe et d'éviter que l'appelant ne transmette un pointeur nul. Si nous utilisons des références plutôt que des pointeurs, voici ce que devient la fonction :

```
double manhattanDistance(const Point2D &a, const Point2D &b)
{
   return abs(b.x() - a.x()) + abs(b.y() - a.y());
}
```

La déclaration d'une référence est analogue à celle d'un pointeur, `&` remplaçant `*`. Mais lorsque nous utilisons effectivement la référence, nous pouvons oublier qu'il s'agit d'une adresse mémoire et la traiter comme une variable ordinaire. De plus, l'appel d'une fonction qui reçoit des références en arguments n'exige aucune attention particulière (pas d'opérateur `&`).

Dans l'ensemble, en remplaçant `Point2D` par `const Point2D &` dans la liste de paramètres, nous avons réduit la surcharge de traitement de l'appel de fonction : plutôt que de copier 256 bits (la taille de quatre `double`), nous copions seulement 64 ou 128 bits, selon la taille du pointeur de la plate-forme cible.

L'exemple précédent s'appuyait sur les références `const`, ce qui empêchait la fonction de modifier les objets associés aux références. Lorsqu'on veut au contraire autoriser ce comportement, nous pouvons transmettre une référence ou un pointeur non-const. Par exemple :

```
void transpose(Point2D &point)
{
    double oldX = point.x();
    point.setX(point.y());
    point.setY(oldX);
}
```

Dans certains cas, nous disposons d'une référence et nous avons besoin d'appeler une fonction qui reçoit un pointeur, ou vice versa. Pour convertir une référence en pointeur, nous pouvons tout simplement utiliser l'opérateur unaire `&` :

```
Point2D point;
Point2D &ref = point;
Point2D *ptr = &ref;
```

Pour convertir un pointeur en référence, il y a l'opérateur unaire `*` :

```
Point2D point;
Point2D *ptr = &point;
Point2D &ref = *ptr;
```

Références et pointeurs sont représentés de la même façon en mémoire, et ils peuvent souvent être utilisés de façon interchangeable, ce qui soulève la question de quel élément à utiliser et quand. D'un côté, la syntaxe des références est plus pratique. D'un autre côté, à tout moment vous pouvez réaffecter les pointeurs pour pointer sur un autre objet, ils peuvent stocker une valeur nulle, et leur syntaxe plus explicite est plutôt une bonne chose. C'est pour cette raison que les pointeurs ont tendance à l'emporter, les références étant plutôt presque exclusivement employée pour déclarer les paramètres de fonction, en conjonction avec `const`.

Les tableaux

Les tableaux en C++ sont déclarés en spécifiant leur nombre d'éléments entre crochets dans la déclaration de variable *après* le nom de variable. Il est possible de créer des tableaux à deux dimensions à l'aide d'un tableau de tableau. Voici la définition d'un tableau à une dimension contenant 10 éléments de type `int` :

```
int fibonacci[10];
```

On accède aux éléments via la syntaxe `fibonacci[0]`, `fibonacci[1]`, …, `fibonacci[9]`. Nous avons souvent besoin d'initialiser le tableau lors de sa définition :

```
int fibonacci[10] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
```

Dans ce cas, nous pouvons alors omettre la taille de tableau puisque le compilateur peut la déduire à partir du nombre d'initialisations :

```
int fibonacci[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
```

L'initialisation statique fonctionne aussi pour les types complexes, tels que `Point2D` :

```
Point2D triangle[] = {
    Point2D(0.0, 0.0), Point2D(1.0, 0.0), Point2D(0.5, 0.866)
};
```

Si nous n'avons pas l'intention de modifier le tableau par la suite, nous pouvons le rendre `const` :

```
const int fibonacci[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
```

Pour trouver le nombre d'éléments d'un tableau, il suffit d'exécuter l'opérateur `sizeof()` comme suit :

```
int n = sizeof(fibonacci) / sizeof(fibonacci[0]);
```

Cet opérateur renvoie la taille de ses arguments en octets. Le nombre d'éléments dans un tableau correspond à la taille de ce dernier en octets divisée par la taille d'un de ses éléments. Comme c'est un peu fatiguant à taper, une solution courante consiste à déclarer une constante et à l'utiliser pour définir le tableau :

```
enum { NFibonacci = 10 };
const int fibonacci[NFibonacci] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
```

Il aurait été tentant de déclarer la constante sous forme de variable `const int`.

Malheureusement, certains compilateurs acceptent mal les variables `const` pour spécifier les tailles de tableau. Le mot-clé `enum` sera expliqué plus tard dans cette annexe. Le parcours d'un tableau est normalement réalisé à l'aide d'un entier. Par exemple :

```
for (int i = 0; i < NFibonacci; ++i)
    cout << fibonacci[i] << endl;
```

Il est aussi possible de parcourir le tableau à l'aide d'un pointeur :

```
const int *ptr = &fibonacci[0];
while (ptr != &fibonacci[10]) {
    cout << *ptr << endl;
    ++ptr;
}
```

Nous initialisons le pointeur avec l'adresse du premier élément et nous bouclons jusqu'à ce que nous atteignons l'élément "qui suit le dernier" (le "onzième" élément, `fibonacci[10]`). A chaque itération, l'opérateur ++ positionne le pointeur sur l'élément suivant.

Au lieu de `&fibonacci[0]`, nous aurions aussi pu écrire `fibonacci`. En effet, le nom d'un tableau utilisé seul est automatiquement converti en pointeur vers le premier élément du tableau. De la même façon, nous pourrions remplacer `fibonacci + 10` par `&fibonacci[10]`. Cela marche aussi en sens inverse : nous pouvons récupérer le contenu de l'élément courant en codant soit `*ptr` soit ptr[0] et nous pourrions accéder à l'élément suivant à l'aide de `*(ptr + 1)` ou `ptr[1]`.

On fait quelquefois référence à ce principe en termes "d'équivalence entre pointeurs et tableaux".

Pour empêcher ce qu'il considère comme une perte d'efficacité gratuite, C++ nous interdit de transmettre des tableaux à des fonctions par valeur. Il faut plutôt transmettre leur adresse. Par exemple :

```
#include <iostream>

using namespace std;

void printIntegerTable(const int *table, int size)
{
    for (int i = 0; i < size; ++i)
    cout << table[i] << endl;
}
```

```
int main()
{
    const int fibonacci[10] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
    printIntegerTable(fibonacci, 10);
    return 0;
}
```

Ironiquement, alors que le C++ ne nous donne aucun choix concernant le mode de transmission d'un tableau, par adresse ou par valeur, il nous accorde quelques libertés dans la *syntaxe* employée pour déclarer le type de paramètre. Pour remplacer `const int *table`, nous aurions pu coder `const int table[]` pour déclarer un paramètre pointeur-vers-constante-`int`. D'une façon analogue, le paramètre `argv` de `main()` peut être déclaré soit en tant que `char *argv[]` soit en tant que `char **argv`.

Pour copier un tableau dans un autre tableau, une solution consiste à boucler dans le tableau :

```
const int fibonacci[NFibonacci] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
int temp[NFibonacci];

for (int i = 0; i < NFibonacci; ++i)
    temp[i] = fibonacci[i];
```

Pour les types de base tel que `int`, nous pouvons aussi utiliser `std::memcpy()`, qui copie un bloc de mémoire. Par exemple :

```
memcpy(temp, fibonacci, sizeof(fibonacci));
```

Lorsque nous déclarons un tableau C++, la taille doit être une constante. Si nous désirons créer un tableau de taille variable, nous avons plusieurs solutions.

- **Nous pouvons allouer dynamiquement ce tableau :**

```
int *fibonacci = new int[n];
```

Certains compilateurs autorisent les variables dans ce contexte, mais il ne faut pas exploiter ce type de fonctionnalité pour créer un programme portable.

L'opérateur `new[]` alloue un certain nombre d'éléments dans des emplacements mémoire consécutifs et renvoie un pointeur vers le premier élément. Grâce au principe de "l'équivalence des pointeurs et des tableaux," on peut accéder aux éléments par l'intermédiaire d'un pointeur avec `fibonacci[0]`, `fibonacci[1]`, …, `fibonacci[n -1]`. Dès que le tableau n'est plus nécessaire, nous devons libérer la mémoire qu'il occupe en exécutant l'opérateur `delete[] :`

```
delete [] fibonacci;
```

- **Nous pouvons appliquer la syntaxe standard** `std::vector<T> class :`

```
#include <vector>

using namespace std;

vector<int> fibonacci(n);
```

Vous accédez aux éléments à l'aide de l'opérateur `[]`, exactement comme dans un tableau C++ ordinaire. Avec `std::vector<T>` (où `T` est le type des éléments stockés dans le vecteur), nous pouvons redimensionner le tableau à tout moment en exécutant `resize()` et nous le copions avec l'opérateur d'affectation. Les classes qui contiennent des crochets (<>) dans leur nom sont appelées classes modèle.

- **Nous pouvons utiliser la syntaxe `QVector<T> class` de Qt :**

```
#include <QVector>

QVector<int> fibonacci(n);
```

L'API de `QVector<T>` est très proche de celle de `std::vector<T>`, mais elle prend aussi en charge l'itération *via* le mot-clé `foreach` de Qt et elle utilise le partage de données ("copie à l'écriture") afin d'optimiser la mémoire et les temps de réponse. Le Chapitre 11 présente les classes conteneur de Qt et expose leurs relations avec les conteneurs standard du C++.

Vous pourriez être tenté d'éviter au maximum les tableaux intégrés et de coder plutôt `std::vector<T>` ou `QVector<T>`. Il est pourtant essentiel de bien comprendre le fonctionnement des tableaux intégrés parce que tôt ou tard vous en aurez besoin dans du code fortement optimisé ou pour servir d'interface avec les bibliothèques C existantes.

Les chaînes de caractères

La façon la plus simple de représenter des chaînes de caractères en C++ consiste à utiliser un tableau de caractères terminé par un octet nul ('0'). Les quatre fonctions suivantes illustrent le comportement de ce type de chaînes :

```
void hello1()
{
   const char str[] = {
     'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'
   };
   cout << str << endl;
}

void hello2()
{
   const char str[] = "Hello world!";
   cout << str << endl;
}

void hello3()
{
   cout << "Hello world!" << endl;
}
```

```
void hello4()
{
   const char *str = "Hello world!";
   cout << str << endl;
}
```

Dans la première fonction, nous déclarons la chaîne en tant que tableau et nous l'initialisons de façon basique. Notez le caractère 0 (zéro) de fin, qui signale la fin de la chaîne. La deuxième fonction comporte une définition de tableau similaire mais nous faisons cette fois appel à un littéral chaîne pour initialiser le tableau. En C++, les littéraux chaîne sont de simples tableaux de `const char` avec un caractère 0 de fin implicite. La troisième fonction emploie directement un littéral chaîne, sans lui donner de nom. Une fois traduit en instructions de langage machine, ce code est identique à celui des deux fonctions précédentes.

La quatrième fonction est un peu différente parce qu'elle crée non seulement un tableau (anonyme) mais aussi une variable pointeur appelée `str` qui stocke l'adresse du premier élément du tableau. Malgré cela, la sémantique de la fonction est identique aux trois fonctions précédentes, et un compilateur performant devrait éliminer la variable `str` superflue.

Les fonctions qui reçoivent des chaînes C++ en arguments reçoivent généralement un `char*` ou un `const char*`. Voici un programme court qui illustre l'emploi de chacune d'elles :

```
#include <cctype>
#include <iostream>

using namespace std;

void makeUppercase(char *str)
{
    for (int i = 0; str[i] != '\0'; ++i)
       str[i] = toupper(str[i]);
}

void writeLine(const char *str)
{
    cout << str << endl;
}

int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; ++i) {
    makeUppercase(argv[i]);
    writeLine(argv[i]);
    }
    return 0;
}
```

En C++, le type `char` stocke normalement une valeur 8 bits. Cela signifie que nous pouvons facilement stocker des chaînes ASCII, ISO 8859-1 (Latin-1), et autres codages 8 bits dans un tableau de caractères, mais que nous ne pouvons pas stocker des caractères Unicode arbitraires sans faire appel à des séquences multioctet. Qt fournit la puissante classe `QString`, qui stocke

des chaînes Unicode sous forme de séquences de `QChars` 16 bits et qui exploite en interne l'optimisation du partage de données implicite ("copie à l'écriture"). Cette classe est détaillée dans les Chapitres 11 (Classes conteneur) et 17 (Internationalisation).

Les énumérations

C++ comprend une fonctionnalité d'énumération pour la déclaration d'un jeu de constantes nommées analogue à celle fournie par C#. Supposons que nous voulions stocker les jours de la semaine dans un programme :

```
enum DayOfWeek {
   Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
};
```

Normalement, nous devrions placer cette déclaration dans un fichier d'en-tête ou même dans une classe. La déclaration précédente est équivalente en apparence aux définitions de constantes suivantes :

```
const int Sunday = 0;
const int Monday = 1;
const int Tuesday = 2;
const int Wednesday = 3;
const int Thursday = 4;
const int Friday = 5;
const int Saturday = 6;
```

En exploitant la construction de l'énumération, nous pouvons déclarer par la suite des variables ou paramètres de type `DayOfWeek` et le compilateur garantira que seules les valeurs de l'énumération `DayOfWeek` leur seront affectées. Par exemple :

```
DayOfWeek day = Sunday;
```

Si la sécurité des types ne nous concerne pas, nous pouvons aussi coder

```
int day = Sunday;
```

Notez que pour faire référence à la constante `Sunday` dans l'énumération `DayOfWeek`, nous écrivons simplement `Sunday`, et non `DayOfWeek::Sunday`.

Par défaut, le compilateur affecte des valeurs d'entiers consécutifs aux constantes d'une énumération, en commençant à 0. Nous pouvons spécifier d'autres valeurs si nécessaire :

```
enum DayOfWeek {
    Sunday = 628,
    Monday = 616,
    Tuesday = 735,
    Wednesday = 932,
    Thursday = 852,
    Friday = 607,
    Saturday = 845
};
```

Si nous ne spécifions pas la valeur d'un élément de l'énumération, celui-ci prend la valeur de l'élément précédent, plus 1. Les énumérations sont quelquefois employées pour déclarer des constantes entières, auquel cas nous omettons normalement le nom de l'énumération :

```
Enum
    FirstPort = 1024,
    MaxPorts = 32767
};
```

Un autre emploi fréquent des énumérations concerne la représentation des ensembles d'options. Prenons l'exemple d'une boîte de dialogue Find, avec quatre cases à cocher contrôlant l'algorithme de recherche (`Wildcard syntax`, `Case sensitive`, `Search backward`, et `Wrap around`). Nous pouvons représenter ces cases à l'aide d'une énumération dans laquelle les constantes sont des puissances de 2 :

```
enum FindOption {
    NoOptions = 0x00000000,
    WildcardSyntax = 0x00000001,
    CaseSensitive = 0x00000002,
    SearchBackward = 0x00000004,
    WrapAround = 0x00000008
};
```

Chaque option est souvent nommée "indicateur". Nous pouvons combiner ces indicateurs à l'aide des opérateurs bit à bit `|` ou `|= :`

```
int options = NoOptions;
if (wilcardSyntaxCheckBox->isChecked())
    options |= WildcardSyntax;
if (caseSensitiveCheckBox->isChecked())
    options |= CaseSensitive;
if (searchBackwardCheckBox->isChecked())
    options |= SearchBackwardSyntax;
if (wrapAroundCheckBox->isChecked())
    options |= WrapAround;
Nous pouvons tester si un indicateur existe à l'aide de l'opérateur & bit à bit:
if (options & CaseSensitive) {
    // rechercher avec casse significative
}
```

Une variable de type `FindOption` ne peut contenir qu'un seul indicateur à la fois. La combinaison de plusieurs indicateurs à l'aide de `|` donne un entier ordinaire. Dans ce cas, les types ne sont malheureusement pas sécurisés : le compilateur ne va rien signaler si une fonction qui doit recevoir une combinaison de `FindOptions` dans un paramètre `int` reçoit plutôt `Saturday`. Qt utilise `QFlags<T>` pour sécuriser les types de ses propres types d'indicateur. La classe est également disponible lorsque nous définissons des types d'indicateur personnalisés. Consultez la documentation en ligne de `QFlags<T>` pour connaître tous les détails.

TypeDef

Avec C++, nous pouvons définir un alias pour un type de données à l'aide du mot-clé `typedef`. Si nous codons souvent par exemple `QVector<Point2D>` et que nous désirons économiser les opérations de frappe, nous pouvons placer la déclaration de `typedef` suivante dans un de nos fichiers d'en-tête :

```
typedef QVector<Point2D> PointVector;
```

A partir de là, `PointVector` peut être codé en lieu et place de `QVector<Point2D>`. Notez que le nouveau nom du type apparaît après l'ancien. La syntaxe de `typedef` suit délibérément celle des déclarations de variable.

Avec Qt, les typedef sont surtout employés pour les raisons suivantes :

- *L'aspect pratique :* Qt déclare `uint` et `QWidgetList` en tant que typedef pour `unsigned int` et `QList<QWidget *>` afin de simplifier la frappe.
- *Les différences entre plates-formes :* certains types ont des définitions différentes sur des plates-formes différentes. Par exemple, `qlonglong` est défini comme `__int64` sous Windows et comme `long long` sur d'autres plates-formes.
- *La compatibilité :* la classe `QIconSet` de Qt 3 a été renommée en `QIcon` dans Qt 4. Pour aider les utilisateurs de Qt 3 à porter leurs applications vers Qt 4, `QIconSet` est fourni comme `typedef` pour `QIcon` lorsque la compatibilité Qt 3 est activée.

Conversions de type

C++ fournit plusieurs syntaxes pour la conversion des valeurs d'un type à l'autre. La syntaxe traditionnelle, héritée de C, implique de placer le type obtenu entre parenthèses avant la valeur à convertir :

```
const double Pi = 3.14159265359;
int x = (int)(Pi * 100);
cout << x << " equals 314" << endl;
```

Cette syntaxe est très puissante. Elle permet de changer le type des pointeurs, de supprimer `const`, et plus encore. Par exemple :

```
short j = 0x1234;
if (*(char *)&j == 0x12)
    cout << "The byte order is big-endian" << endl;
```

Dans l'exemple précédent, nous convertissons un `short*` en `char*` et nous utilisons l'opérateur unaire * pour accéder à l'octet situé à l'emplacement de mémoire donné. Sur les systèmes big-endian, cet octet se trouve en `0x12` ; sur les systèmes little-endian, il se trouve à l'emplacement `0x34`. Les pointeurs et références étant représentés de la même façon, vous ne serez pas surpris d'apprendre que le code précédent peut être réécrit à l'aide d'une conversion de référence :

```
short j = 0x1234;
if ((char &)j == 0x12)
    cout << "The byte order is big-endian" << endl;
```

Si le type de données est un nom de classe, un typedef, ou un type primitif qui peut s'exprimer sous la forme d'un unique jeton alphanumérique, nous pouvons nous servir de la syntaxe du constructeur comme d'une conversion :

```
int x = int(Pi * 100);
```

La conversion des pointeurs et références à l'aide des conversions traditionnelles de style C est assez périlleuse parce que le compilateur nous autorise à convertir n'importe quel type de pointeur (ou référence) vers n'importe quel autre type de pointeur (ou référence). C'est pourquoi C++ a introduit quatre conversions de style nouveau avec une sémantique plus précise. Pour les pointeurs et références, les nouvelles conversions sont préférables aux conversions de style C plus risquées et sont utilisées dans cet ouvrage.

- `static_cast<T>()` convertit un pointeur-vers-A en pointeur-vers-B, en imposant que la classe B hérite de la classe A. Par exemple :

  ```
  A *obj = new B;
  B *b = static_cast<B *>(obj);
  b->someFunctionDeclaredInB();
  ```

  Si l'objet n'est pas une instance de B (mais qu'il hérite de A), l'usage du pointeur obtenu peut produire des pannes obscures.

- `dynamic_cast<T>()` est analogue à `static_cast<T>()`, sauf que les informations de type à l'exécution (RTTI) servent à contrôler si l'objet associé au pointeur est bien une instance de la classe B. Si ce n'est pas le cas, la conversion renvoie un pointeur nul. Par exemple :

  ```
  A *obj = new B;
  B *b = dynamic_cast<B *>(obj);
  if (b)
     b-> someFunctionDeclaredInB();
  ```

  Avec certains compilateurs, `dynamic_cast<T>()` ne fonctionne pas au-delà des limites de la bibliothèque dynamique. Cet élément s'appuie également sur la prise en charge de RTTI par le compilateur, une fonctionnalité que tout bon programmeur doit désactiver pour réduire la taille des exécutables. Qt résout ces problèmes en fournissant `qobject_cast<T>()` pour les sous-classes de `QObject`.

- `const_cast<T>()` ajoute ou supprime un qualificateur `const` sur un pointeur ou une référence. Par exemple :

  ```
  int MyClass::someConstFunction() const
  {
      if (isDirty()) {
         MyClass *that = const_cast<MyClass *>(this);
  ```

```
        that->recomputeInternalData();
    }
    ...
}
```

Dans l'exemple précédent, nous convertissons le qualificateur `const` du pointeur `this` afin d'appeler la fonction membre non-const `recomputeInternalData()`. Il n'est pas recommandé de procéder de cette façon et vous pouvez l'éviter en codant `mutable`, comme expliqué au Chapitre 4.

- `reinterpret_cast<T>()` convertit tout type pointeur ou référence vers un autre type de ce genre. Par exemple :

```
short j = 0x1234;
if (reinterpret_cast<char &>(j) == 0x12)
    cout << "The byte order is big-endian" << endl;
```

En Java et C#, il est possible de stocker toute référence en tant que référence d'`Object`. C++ ne comporte pas une telle classe de base universelle, mais il fournit un type de données particulier, `void*`, qui stocke l'adresse d'une instance de n'importe quel type. Un `void*` doit être reconvertit vers un autre type (à l'aide de `static_cast<T>()`) avant d'être exploité.

C++ offre de nombreuses méthodes de conversion entre types, mais nous avons rarement besoin d'effectuer une conversion dans ce langage. Lorsque nous utilisons des classes conteneurs telles que `std::vector<T>` ou `QVector<T>`, nous spécifions le type `T` et nous récupérons les éléments sans convertir. De plus, pour les types primitifs, certaines conversions s'effectuent implicitement (de `char` vers `int` par exemple), et pour les types personnalisés nous pouvons définir des conversions implicites en fournissant un constructeur à un paramètre. Par exemple :

```
class MyInteger
{
Public
    MyInteger();
    MyInteger(int i);
    ...
};

int main()
{
    MyInteger n;
    n = 5;
    ...
}
```

Pour certains constructeurs à un paramètre, la conversion automatique n'est pas justifiée. Il est possible de la désactiver en déclarant le constructeur avec le mot-clé `explicit` :

```
class MyVector
{
```

```
Public
    explicit MyVector(int size);
    ...
};
```

Surcharge d'opérateur

C++ nous permet de surcharger des fonctions, ce qui signifie que nous pouvons déclarer plusieurs fonctions avec le même nom dans la même portée, à partir du moment où leurs listes de paramètres sont différentes. C++ prend aussi en charge la *surcharge des opérateurs*. Il s'agit de la possibilité d'affecter une sémantique particulière à des opérateurs intégrés (tels que +, <<, et `[]`) lorsqu'ils sont employés avec des types personnalisés.

Nous avons déjà étudié quelques exemples d'opérateurs surchargés. Lorsque nous avions utilisé << pour transmettre du texte en sortie vers `cout` ou `cerr`, nous n'avions pas exécuté l'opérateur de décalage de C++, mais plutôt une version particulière de cet opérateur qui recevait un objet `ostream` (tel que `cout` et `cerr`) à gauche et une chaîne (ou bien un nombre ou un manipulateur de flux tel que `endl`) à droite et qui renvoyait l'objet `ostream`, ce qui permettait d'effectuer plusieurs appels par ligne.

L'intérêt de la surcharge des opérateurs est que nous pouvons obtenir un comportement pour nos types personnalisés identique à celui des types intégrés. Pour illustrer ce point, nous allons surcharger +=, -=, +, et - pour travailler avec des objets `Point2D :`

```
#ifndef POINT2D_H
#define POINT2D_H

class Point2D
{
Public
   Point2D();
   Point2D(double x, double y);

   void setX(double x);
   void setY(double y);
   double x() const;
   double y() const;

   Point2D &operator+=(const Point2D &other) {
      xVal += other.xVal;
      yVal += other.yVal;
      return *this;
   }
   Point2D &operator-=(const Point2D &other) {
      xVal -= other.xVal;
      yVal -= other.yVal;
      return *this;
   }

Private
```

```
   double xVal;
   double yVal;
};

inline Point2D operator+(const Point2D &a, const Point2D &b)
{
   return Point2D(a.x() + b.x(), a.y() + b.y());
}

inline Point2D operator-(const Point2D &a, const Point2D &b)
{
   return Point2D(a.x() - b.x(), a.y() - b.y());
}

#endif
```

Les opérateurs peuvent être implémentés soit en tant que fonctions membres soit en tant que fonctions globales. Dans notre exemple, nous avons implémenté `+=` et `-=` en tant que fonctions membres, et `+` et `-` en tant que fonctions globales.

Les opérateurs `+=` et `-=` reçoivent la référence d'un autre objet `Point2D` et incrémentent ou décrémentent les coordonnées *x* et *y* de l'objet courant en fonction de l'autre objet. Ils renvoient `*this`, qui représente une référence à l'objet courant (de type `Point2D*`). Puisqu'on renvoie une référence, nous pouvons écrire du code un peu exotique comme ci-après :

```
a += b += c;
```

Les opérateurs `+` et `-` reçoivent deux paramètres et renvoient un objet `Point2D` par valeur (et non la référence d'un objet existant). Grâce au mot-clé `inline`, nous avons la possibilité de placer ces définitions de fonction dans le fichier d'en-tête. Si le corps de la fonction avait été plus long, nous aurions inséré le prototype de fonction dans le fichier d'en-tête et la définition de fonction (sans le mot-clé `inline`) dans le fichier `.cpp`.

L'extrait de code suivant présente les quatre opérateurs surchargés en action :

```
Point2D alpha(12.5, 40.0);
Point2D beta(77.5, 50.0);

alpha += beta;
beta -= alpha;

Point2D gamma = alpha + beta;
Point2D delta = beta - alpha;
```

Nous invoquons les fonctions opérateur comme n'importe quelle autre fonction :

```
Point2D alpha(12.5, 40.0);
Point2D beta(77.5, 50.0);

alpha.operator+=(beta);
beta.operator-=(alpha);
```

```
Point2D gamma = operator+(alpha, beta);
Point2D delta = operator-(beta, alpha);
```

La surcharge des opérateurs en C++ est un sujet complexe, mais vous n'avez pas forcément besoin d'en connaître tous les détails. Il est surtout important de comprendre les principes de base de la surcharge des opérateurs parce que plusieurs classes Qt (notamment `QString` et `QVector<T>`) s'appuient sur cette fonctionnalité pour fournir une syntaxe plus simple et plus naturelle pour des opérations telles que la concaténation et l'ajout d'élément.

Les types valeur

Java et C# différencient les types valeur et les types référence.

- *Types valeur* : il s'agit des types primitifs tels que `char`, `int`, et `float`, ainsi que les structures C#. Ils sont caractérisés par le fait qu'ils ne sont pas créés à l'aide de `new` et que l'opérateur d'affectation copie la valeur stockée dans la variable. Par exemple :

```
int i = 5;
int j = 10;
i = j;
```

- *Types référence* : Il s'agit de classes telles que `Integer` (en Java), `String`, et `MaClasseAmoi`. Les instances sont créées en codant `new`. L'opérateur d'affectation copie uniquement une référence de l'objet ; pour obtenir une copie intégrale, il faut appeler `clone()` (en Java) ou `Clone()` (en C#). Par exemple :

```
Integer i = new Integer(5);
Integer j = new Integer(10);
i = j.clone();
```

En C++, tous les types peuvent être employés en tant que "types référence," et ceux qui peuvent être copiés peuvent aussi être employés en tant que "types valeur".C++n'a pas besoin d'un classe `Integer`, par exemple, parce que vous pouvez vous servir des pointeurs et de `new` comme dans les lignes suivantes :

```
int *i = new int(5);
int *j = new int(10);
*i = *j;
```

Contrairement à Java et C#, C++ traite les classes définies par l'utilisateur exactement comme des types intégrés :

```
Point2D *i = new Point2D(5, 5);
Point2D *j = new Point2D(10, 10);
*i = *j;
```

Si nous désirons rendre une classe C++ copiable, nous devons prévoir un constructeur de copie et un opérateur d'affectation pour cette dernière. Le constructeur de copie est invoqué lorsque

nous initialisons l'objet avec un autre objet de même type. C++ fournit deux syntaxes équivalentes pour cette situation :

```
Point2D i(20, 20);
Point2D j(i); // première syntaxe
Point2D k = i; // deuxième syntaxe
```

L'opérateur d'affectation est invoqué lorsque nous utilisons cet opérateur sur une variable existante :

```
Point2D i(5, 5);
Point2D j(10, 10);
j = i;
```

Lorsque nous définissons une classe, le compilateur C++ fournit automatiquement un constructeur de copie et un opérateur d'affectation qui effectue une copie membre à membre. Pour la classe `Point2D`, c'est comme si nous avions écrit le code suivant dans la définition de classe :

```
class Point2D
{
Public
    ...
    Point2D(const Point2D &other)
        : xVal(other.xVal), yVal(other.yVal) { }

    Point2D &operator=(const Point2D &other) {
      xVal = other.xVal;
      yVal = other.yVal;
      return *this;
    }
    ...

Private
    double xVal;
    double yVal;
};
```

Pour certaines classes, le constructeur de copie par défaut et l'opérateur d'affectation ne conviennent pas. Cela se produit en particulier lorsque la classe travaille en mémoire dynamique. Pour la rendre copiable, nous devons alors implémenter nous-mêmes le constructeur de copie et l'opérateur d'affectation.

Pour les classes qui n'ont pas besoin d'être copiables, nous avons la possibilité de désactiver le constructeur de copie et l'opérateur d'affectation en les rendant privés. Si nous tentons accidentellement de copier des instances de cette classe, le compilateur signale une erreur. Par exemple :

```
class BankAccount
{
Public
```

```
    ...

Private
    BankAccount(const BankAccount &other);
    BankAccount &operator=(const BankAccount &other);
};
```

En Qt, beaucoup de classes sont conçues pour être utilisées en tant que classes de valeur. Elles disposent d'un constructeur de copie et d'un opérateur d'affectation, et elles sont normalement instanciées sur la pile sans `new`. C'est le cas en particulier pour `QDateTime`, `QImage`, `QString`, et les classes conteneur telles que `QList<T>`, `QVector<T>`, et `QMap<K,T>`.

Les autres classes appartiennent à la catégorie "type référence", notamment `QObject` et ses sous-classes (`QWidget`, `QTimer`, `QTcpSocket`, etc.). Celles-ci possèdent des fonctions virtuelles et ne peuvent pas être copiées. Un `QWidget` représente par exemple une fenêtre spécifique ou un contrôle sur l'écran. S'il y a 75 instances de `QWidget` en mémoire, il y a aussi 75 fenêtres et contrôles sur cet écran. Ces classes sont typiquement instanciées à l'aide de l'opérateur `new`.

Variables globales et fonctions

C++ autorise la déclaration de fonctions et de variables qui n'appartiennent à aucune classe et qui sont accessibles dans n'importe quelle autre fonction. Nous avons étudié plusieurs exemples de fonctions globales, dont `main()`, le point d'entrée du programme. Les variables globales sont plus rares, parce qu'elles compromettent la modularité et la réentrance du thread. Il est important de bien les comprendre parce que vous pourriez les retrouver dans du code écrit par d'anciens programmeurs C ou d'autres utilisateurs de C++.

Nous allons illustrer le fonctionnement des fonctions et variables globales en étudiant un petit programme qui affiche une liste de 128 nombres pseudo-aléatoires à l'aide d'un algorithme rapide et simple. Le code source du programme est réparti dans deux fichiers `.cpp`.

Le premier est `random.cpp` :

```
int randomNumbers[128];

static int seed = 42;

static int nextRandomNumber()
{
    seed = 1009 + (seed * 2011);
    return seed;
}

void populateRandomArray()
{
    for (int i = 0; i < 128; ++i)
       randomNumbers[i] = nextRandomNumber();
}
```

Dans ce fichier, deux variables globales (`randomNumbers` et `seed`) et deux fonctions globales (`nextRandomNumber()` et `populateRandomArray()`) sont déclarées. Deux déclarations contiennent le mot-clé `static` ; elles ne sont visibles que dans l'unité de compilation courante (`random.cpp`) et on dit qu'elles ont une *liaison statique*. Les deux autres sont disponibles dans n'importe quelle unité de compilation du programme ; elles ont une *liaison externe*.

La liaison statique est particulièrement indiquée pour les fonctions assistantes et les variables internes qui ne doivent pas être utilisées dans d'autres unités de compilation. Elle réduit le risque de collision d'identificateurs (variables globales de même nom ou fonctions globales avec la même signature dans des unités de compilation différentes) et empêche des utilisateurs malveillants ou très maladroits d'accéder au code d'une unité de compilation.

Examinons maintenant le second fichier, `main.cpp`, qui utilise les deux variables globales déclarées avec une liaison externe dans `random.cpp` :

```
#include <iostream>


using namespace std;

extern int randomNumbers[128];

void populateRandomArray();

int main()
{
    populateRandomArray();
    for (int i = 0; i < 128; ++i)
      cout << randomNumbers[i] << endl;
    return 0;
}
```

Nous déclarons les variables et fonctions externes avant de les appeler. La déclaration de variable externe (qui rend la variable visible dans l'unité de compilation courante) pour `randomNumbers` débute avec le mot-clé `extern`. Sans ce mot-clé, le compilateur pourrait penser qu'il s'agit d'une *définition* de variable, et l'éditeur de liens rapporterait une erreur pour la même variable définie dans deux unités de compilation différentes (`random.cpp` et `main.cpp`). Vous pouvez déclarer des variables autant de fois que vous voulez, mais elles ne doivent être définies qu'une seule fois. C'est à partir de la définition que le compilateur réserve l'espace requis pour la variable.

La fonction `populateRandomArray()` est déclarée avec un prototype de fonction. Le mot-clé `extern` est facultatif pour les fonctions.

Nous pourrions typiquement placer les déclarations de fonction et de variable externes dans un fichier d'en-tête et inclure ce dernier dans tous les fichiers qui en ont besoin :

```
#ifndef RANDOM_H
#define RANDOM_H
```

```
extern int randomNumbers[128];

void populateRandomArray();

#endif
```

Nous avions déjà vu comment coder `static` pour déclarer des fonctions et variables membres qui ne sont pas associées à une instance spécifique de la classe, et nous venons maintenant de voir comme utiliser ce mot-clé pour déclarer des fonctions et variables avec une liaison statique. Notez qu'il existe un autre usage de `static`. En C++, nous pouvons déclarer une variable locale avec ce mot-clé. De telles variables sont initialisées la première fois que la fonction est appelée et leur valeur est garantie entre plusieurs appels de fonction. Par exemple :

```
void nextPrime()
{
    static int n = 1;

    do {
       ++n;
    } while (!isPrime(n));

    return n;
}
```

Les variables locales statiques sont similaires aux variables globales, mais elles ne sont visibles qu'à l'intérieur de la fonction dans laquelle elles sont définies.

Espaces de noms

Les espaces de noms sont un mécanisme destiné à réduire les risques de collision de noms dans les programmes C++. Ces collisions sont fréquentes dans les gros programmes qui exploitent plusieurs bibliothèques tiers. Dans vos propres programmes, vous avez le choix de les utiliser ou non.

Nous encadrons typiquement dans un espace de noms toutes les déclarations d'un fichier d'entête afin de ne pas "polluer" l'espace de noms global avec les identificateurs qu'il contient. Par exemple :

```
#ifndef SOFTWAREINC_RANDOM_H
#define SOFTWAREINC_RANDOM_H

namespace SoftwareInc
{
    extern int randomNumbers[128];

    void populateRandomArray();
}

#endif
```

(Notez que nous avons aussi renommé la macro de préprocesseur utilisée pour éviter les inclusions multiples, réduisant ainsi le risque de collision de nom avec un fichier d'en-tête de même nom mais situé dans un répertoire différent.)

La syntaxe d'un espace de noms est similaire à celle d'une classe, mais la ligne ne se termine pas par un point-virgule. Voici le nouveau fichier `random.cpp` :

```
#include "random.h"

int SoftwareInc::randomNumbers[128];

static int seed = 42;

static int nextRandomNumber()
{
    seed = 1009 + (seed * 2011);
    return seed;
}

void SoftwareInc::populateRandomArray()
{
    for (int i = 0; i < 128; ++i)
       randomNumbers[i] = nextRandomNumber();
}
```

Contrairement aux classes, les espaces de noms peuvent être "réouverts" à tout moment. Par exemple :

```
namespace Alpha
{
    void alpha1();
    void alpha2();
}

namespace Beta
{
    void beta1();
}

namespace Alpha
{
    void alpha3();
}
```

Vous avez ainsi la possibilité de définir des centaines de classes, situées dans autant de fichiers d'en-tête, dans un seul espace de noms. La bibliothèque standard C++ englobe ainsi tous ses identificateurs dans l'espace de noms `std`. Dans Qt, les espaces de noms servent pour des identificateurs de type global tels que `Qt::AlignBottom` et `Qt::yellow`. Pour des raisons historiques, les classes de Qt n'appartiennent à aucun espace de noms mais sont préfixées de la lettre "Q".

Pour se référer à un identificateur déclaré dans un espace de noms depuis l'extérieur de cet espace, nous le préfixons avec le nom de l'espace de noms (et `::`). Nous pouvons aussi faire appel à l'un des trois mécanismes suivants, qui visent à réduire la frappe.

- **Définir un alias d'espace de noms :**

```
namespace ElPuebloDeLaReinaDeLosAngeles
{
    void beverlyHills();
    void culverCity();
    void malibu();
    void santaMonica();
}

namespace LA = ElPuebloDeLaReinaDeLosAngeles;
```

  Après la définition de l'alias, celui-ci peut être employé à la place du nom original.

- **Importer un identificateur unique depuis un espace de noms :**

```
int main()
{
    using ElPuebloDeLaReinaDeLosAngeles::beverlyHills;

    beverlyHills();
 ...
}
```

  La déclaration `using` nous permet d'accéder à un identificateur donné dans un espace de noms sans avoir besoin de le préfixer avec le nom de l'espace de noms.

- **Importer un espace de noms complet *via* une unique directive :**

```
int main()
{
    using namespace ElPuebloDeLaReinaDeLosAngeles;
    santaMonica();
    malibu();
    ...
}
```

Cette approche augmente les risques de collision de noms. Dès que le compilateur signale un nom ambigu (deux classes de même nom, par exemple, définies dans deux espaces de noms différents), nous sommes toujours en mesure de qualifier l'identificateur avec le nom de l'espace de noms lorsque nous avons besoin d'y faire référence.

Le préprocesseur

Le préprocesseur C++ est un programme qui convertit un fichier source `.cpp` contenant des directives `#-` (telles que `#include`, `#ifndef`, et `#endif`) en fichier source ne contenant aucune directive de cette sorte. Ces directives effectuent des opérations de texte simples sur le

fichier source, comme une compilation conditionnelle, une inclusion de fichier et une expansion de macro. Le préprocesseur est normalement invoqué automatiquement par le compilateur, mais dans la plupart des systèmes vous avez encore la possibilité de l'invoquer seul (souvent par l'intermédiaire d'une option de compilateur `-E` ou `/E`).

- La directive `#include` insère localement le contenu du fichier indiqué entre crochets (<>) ou guillemets doubles (""), selon que le fichier d'en-tête est installé sur un emplacement standard ou avec les fichiers du projet courant. Le nom de fichier peut contenir `..` et `/` (que les compilateurs Windows interprètent correctement comme étant des séparateurs de répertoire). Par exemple :

```
#include "../shared/globaldefs.h"
```

- La directive `#define` définit une macro. Les occurrences de la macro qui apparaissent après la directive `#define` sont remplacées par la définition de la macro. Par exemple, la directive

```
#define PI 3.14159265359
```

demande au préprocesseur de remplacer toutes les futures occurrences du jeton `PI` dans l'unité de compilation courante par le jeton 3.14159265359. Pour éviter les collisions entre noms de variable et de classe, on affecte généralement aux macros des noms en majuscules. Nous pouvons très bien définir des macros qui reçoivent des arguments :

```
#define SQUARE(x) ((x) * (x))
```

Dans le corps de la macro, il est conseillé d'encadrer toutes les occurrences des paramètres par des parenthèses, ainsi que le corps complet, pour éviter les problèmes de priorité des opérateurs. Après tout, nous voulons que `7*SQUARE(2+3)` soit développé en `7*((2+3)*(2+3))`, et non en `7*2+3*2+3`.

Les compilateurs C++ nous permettent normalement de définir des macros sur la ligne de commande, *via* l'option `-D` ou `/D`. Par exemple :

```
CC -DPI=3.14159265359 -c main.cpp
```

Les macros étaient très populaires avant l'arrivée des typedef, énumérations, constantes, fonctions inline et modèles. Leur rôle aujourd'hui consiste surtout à protéger les fichiers d'en-tête contre les inclusions multiple.

- Vous annulez la définition d'une macro à tout moment à l'aide de `#undef :`

```
#undef PI
```

  C'est pratique si nous avons l'intention de redéfinir une macro, puisque le préprocesseur interdit de définir deux fois la même macro. Ça sert aussi à contrôler la compilation conditionnelle.

- Vous traitez des parties du code ou vous les sautez en codant `#if`, `#elif`, `#else`, et `#endif`, en fonction de la valeur numérique des macros. Par exemple :

```
#define NO_OPTIM 0
#define OPTIM_FOR_SPEED 1
#define OPTIM_FOR_MEMORY 2

#define OPTIMIZATION OPTIM_FOR_MEMORY

...

#if OPTIMIZATION == OPTIM_FOR_SPEED
typedef int MyInt;
#elif OPTIMIZATION == OPTIM_FOR_MEMORY
typedef short MyInt;
#else
typedef long long MyInt;
#endif
```

Dans l'exemple ci-dessus, seule la deuxième déclaration de `typedef` devra être traitée par le compilateur, `MyInt` étant ainsi défini comme un synonyme de `short`. En changeant la définition de la macro OPTIMIZATION, nous obtenons des programmes différents. Lorsqu'une macro n'est pas définie, sa valeur est considérée comme étant 0.

Une autre approche de la compilation conditionnelle consiste à tester si une macro est définie ou non. Vous procédez en codant l'opérateur `defined()` comme suit :

```
#define OPTIM_FOR_MEMORY

...

#if defined(OPTIM_FOR_SPEED)
typedef int MyInt;
#elif defined(OPTIM_FOR_MEMORY)
typedef short MyInt;
#else
typedef long long MyInt;
#endif
```

- Pour des raisons pratiques, le préprocesseur reconnaît `#ifdef X` et `#ifndef X` comme des synonymes de `#if defined(X)` et `#if !defined(X)`. Vous protégez un fichier d'en-tête des inclusions multiple en encadrant son contenu comme ci-après :

```
#ifndef MYHEADERFILE_H
#define MYHEADERFILE_H

...

#endif
```

La première fois que le fichier d'en-tête est inclus, le symbole `MYHEADERFILE_H` n'est pas défini, donc le compilateur traite le code entre `#ifndef` et `#endif`. La seconde et toutes les fois suivantes où le fichier d'en-tête est inclus, `MYHEADERFILE_ H` est déjà défini, donc le bloc `#ifndef … #endif` complet est ignoré.

- La directive `#error` émet un message d'erreur défini par l'utilisateur au moment de la compilation. Elle est souvent associée à une compilation conditionnelle pour signaler un cas impossible. Par exemple :

```
class UniChar
{
Public
#if BYTE_ORDER == BIG_ENDIAN
    uchar row;
    uchar cell;
#elif BYTE_ORDER == LITTLE_ENDIAN
    uchar cell;
    uchar row;
#else
#error "BYTE_ORDER must be BIG_ENDIAN or LITTLE_ENDIAN"
#endif
};
```

Contrairement à la plupart des autres constructions C++, pour lesquelles les espaces n'étaient pas significatifs, les directives de préprocesseur doivent apparaître sur leur propre ligne, sans point-virgule à la fin. Les directives très longues peuvent occuper plusieurs lignes en terminant chacune d'elles sauf la dernière par un slash inversé (\).

La bibliothèque C++ standard

Dans cette section, nous allons rapidement passer en revue la bibliothèque C++ standard. La Figure B.3 répertorie les principaux fichiers d'en-tête C++. Les en-têtes `<exception>`, `<limits>`, `<new>`, et `<typeinfo>` prennent en charge le langage C++ ; par exemple, `<limits>` nous permet de tester les propriétés du support arithmétique des types entier et virgule flottante sur le compilateur, et `<typeinfo>` offre une introspection de base. Les autres en-têtes fournissent généralement des classes pratiques, notamment une classe chaîne et un type numérique complexe. La fonctionnalité offerte par `<bitset>`, `<locale>`, `<string>`, et `<typeinfo>` reprend largement celle des classes `QBitArray`, `QLocale`, `QString`, et `QMetaObject` de Qt.

Le C++ standard inclut aussi un ensemble de fichiers d'en-tête qui traitent les E/S, énumérés en Figure B.4. La conception des classes d'entrées/sorties standard datant des années 80, elles sont assez complexes et donc difficiles à étendre. L'opération est d'ailleurs tellement complexe que des livres complets ont été écrits sur le sujet. Le programmeur hérite d'ailleurs avec ces classes d'un bon nombre de problèmes non résolus liés au codage des caractères et aux représentations binaires dépendantes de la plate-forme des types de données primitifs.

Fichier d'en-tête	*Description*
`<bitset>`	Classe modèle pour représenter des séquences de bits de longueur fixe
`<complex>`	Classe modèle pour représenter des nombres complexes
`<exception>`	Types et fonctions liés à la gestion des exceptions
`<limits>`	Classe modèle qui spécifie les propriétés des types numériques
`<locale>`	Classes et fonctions liées à la localisation
`<new>`	Fonctions qui gèrent l'allocation de mémoire dynamique
`<stdexcept>`	Types d'exception prédéfinis pour signaler les erreurs
`<string>`	Conteneur de chaînes modèle et caractéristiques des caractères
`<typeinfo>`	Classe qui fournit des informations de base concernant un type
`<valarray>`	Classes modèle pour représenter des tableaux de valeurs

Figure B.3
Principaux fichiers d'en-tête de la bibliothèque C++

Fichier d'en-tête	*Description*
`<fstream>`	Classes modèle qui manipulent des fichiers externes
`<iomanip>`	Manipulateur de flux d'E/S qui reçoit un argument
`<ios>`	Classe de base modèle pour les flux d'E/S
`<iosfwd>`	Déclarations anticipées pour plusieurs classes modèles de flux d'E/S
`<iostream>`	Flux d'E/S standard (`cin`, `cout`, `cerr`, `clog`)
`<istream>`	Classe modèle qui contrôle les entrées en provenance d'une mémoire tampon de flux
`<ostream>`	Classe modèle qui contrôle les sorties vers une mémoire tampon de flux
`<sstream>`	Classes modèle qui associent des mémoires tampon de flux avec des chaînes
`<streambuf>`	Classes modèle qui placent en mémoire tampon les opérations d'E/S
`<strstream>`	Classes pour effectuer des opérations de flux d'E/S sur des tableaux de caractères

Figure B.4
Fichiers d'en-tête de la bibliothèque d'E/S C++

Le Chapitre 12 consacré aux entrées/sorties détaille les classes Qt correspondantes, qui présentent les entrées/sorties Unicode ainsi qu'un jeu important de codages de caractères nationaux et une abstraction indépendante de la plate-forme pour stocker les données binaires. Les classes d'E/S de Qt forment la base des communications interprocessus, de la gestion de réseau, et du support XML de Qt. Les classes de flux de texte et binaire de Qt sont très faciles à étendre pour prendre en charge les types de données personnalisés.

La bibliothèque STL (*Standard Template Library*) existe depuis les années 90 et propose un jeu de classes conteneur, itérateurs et algorithmes basés sur des modèles, qui font maintenant partie de la norme standard ISO C++.

La Figure B.5 énumère les fichiers d'en-tête de la STL. La conception de cette bibliothèque est rigoureuse, presque mathématique, et elle fournit une fonctionnalité générique sécurisée au niveau des types. Qt fournit ses propres classes conteneur, dont la conception s'inspire de celle de la STL. Ces classes sont détaillées au Chapitre 11.

Fichier d'en-tête	*Description*
`<algorithm>`	Fonctions modèle à usage général
`<deque>`	Conteneur modèle de file d'attente à double accès
`<functional>`	Modèles d'aide à la construction et manipulation des functeurs (objets fonction)
`<iterator>`	Modèles d'aide à la construction et manipulation des itérateurs
`<list>`	Conteneur modèle de listes doublement chaînées
`<map>`	Conteneurs modèle de map à valeur unique ou multiple
`<memory>`	Utilitaires pour simplifier la gestion de mémoire
`<numeric>`	Opérations numériques modèles
`<queue>`	Conteneur modèle de files d'attente
`<set>`	Conteneurs modèle d'ensembles à valeur unique ou multiple
`<stack>`	Conteneur modèle de piles
`<utility>`	Fonctions modèle de base
`<vector>`	Conteneur modèle de vecteurs

Figure B.5
fichiers d'en-tête de la STL

C++ étant surtout une version améliorée du langage de programmation C, les programmeurs C++ disposent aussi de la bibliothèque C complète. Les fichiers d'en-tête C sont disponibles

soit sous leurs noms traditionnels (par exemple `<stdio.h>`) soit sous leur forme moderne avec le préfixe `c-` et sans l'extension `.h` (`<cstdio>`, par exemple). Lorsque nous codons avec le nom moderne, les fonctions et types de données sont déclarés dans l'espace de noms `std`. (Cela ne s'applique pas à des macros telles que `ASSERT()`, parce que le préprocesseur ne connaît pas les espaces de noms.) La nouvelle syntaxe est recommandée si votre compilateur la prend en charge.

La Figure B.6 répertorie les fichiers d'en-tête C. La plupart proposent une fonctionnalité qui recouvre celle de fichiers d'en-tête C++ ou de Qt plus récents. Notez que `<cmath>` est une exception, ce fichier déclare des fonctions mathématiques telles que `sin()`, `sqrt()`, et `pow()`.

Fichier d'en-tête	*Description*
`<cassert>`	La macro ASSERT()
`<cctype>`	Fonctions pour classer et faire correspondre les caractères
`<cerrno>`	Macros liées au signalement des conditions d'erreur
`<cfloat>`	Macros spécifiant les propriétés des types virgule flottante primitifs
`<ciso646>`	Autres orthographes pour les utilisateurs du jeu de caractères ISO 646
`<climits>`	Macros spécifiant les propriétés des types entiers primitifs
`<clocale>`	Fonctions et types liés à la localisation
`<cmath>`	Fonctions et constantes mathématiques
`<csetjmp>`	Fonctions pour exécuter des branchements non locaux
`<csignal>`	Fonctions pour gérer les signaux du système
`<cstdarg>`	Macros pour implémenter les fonctions à liste d'argument variable
`<cstddef>`	Définitions courantes de plusieurs en-têtes standard
`<cstdio>`	Fonctions pour effectuer les E/S
`<cstdlib>`	Fonctions utilitaires générales
`<cstring>`	Fonctions pour manipuler les tableaux de caractères
`<ctime>`	Types et fonctions pour manipuler le temps
`<cwchar>`	Fonctions de manipulation de chaînes de caractères étendus
`<cwctype>`	Fonctions pour classer et faire correspondre les caractères étendus

Figure B.6
Fichiers d'en-tête C++ pour les utilitaires de bibliothèque C

Cette rapide présentation de la bibliothèque C++ standard s'achève à présent. Dinkumware propose sur Internet une documentation de référence complète concernant la bibliothèque C++ standard à l'adresse **http://www.dinkumware.com/refxcpp.html**, et SGI offre un guide du programmeur STL à l'adresse **http://www.sgi.com/tech/stl/**. La définition officielle de cette bibliothèque se trouve avec les standard C et C++, disponibles sous forme de fichiers PDF ou de copies papier à demander auprès de l'organisation ISO (*International Organization for Standardization*).

Cette annexe traite en très peu de pages un sujet normalement fort étendu. Quand vous allez commencer à étudier Qt à partir du Chapitre 1, vous allez découvrir que la syntaxe est beaucoup plus simple que cette annexe ne pouvait le laisser supposer. Vous pouvez très bien programmer en Qt en faisant uniquement appel à un sous-ensemble de C++ et sans avoir besoin de la syntaxe plus complexe que vous pouvez retrouver dans ce langage. Dès que vous aurez commencé à saisir du code et à créer et lancer vos exécutables, vous ne pourrez que constater à quel point l'approche de Qt est simple et claire. Et dès que vous aborderez des programmes plus ambitieux, en particulier ceux qui ont besoin d'un graphisme performant et élaboré, la combinaison C++/Qt continuera à vous donner toute satisfaction.

Index

Symboles

#define 487, 521
#endif 445, 487
#ifdef 445
#ifndef 487
#include 486, 521
#undef 521
& 18, 497
() 94
<algorithm> 276
<cmath> 125
<iostream> 86
<QtAlgorithms> 276
<QtGui> 18, 48

A

About Qt 53
about() (QMessageBox) 70
accept() 411
    QDialog 31, 36
Acceptable (QSpinBox) 107
acceptProposedAction() 215
acquire() 414, 415
Action 51
    About Qt 53
    Auto-Recalculate 52
    exitAction 53
    New 51
    Open 52
    Save 52
    Save As 52
    Select All 52
    Show Grid 52
activateWindow() 65
Active (groupe de couleurs) 115
activeEditor() 162
ActiveQt 448
ActiveX 460
    sous Windows 448
addAction() (QMenu) 54
addBindValue() 313
addCd() 326
addChildSettings() 235
addDatabase() 311, 314
addItem() (QComboBox) 241
addLibraryPath() 435
addMenu() (QMenu) 54
AddRef() 456
addRow() 233
addStretch() 148
addTrack() 327
addTransaction() 420-421
addWidget() 147, 151
    QStatusBar 57
adjust() 139
    PlotSettings 134
adjustAxis() 139
adjusted() 130
adjustSize() 127
Aide en ligne 373
    Assistant Qt 379
    avec l'assistant 379
    infobulle 374
    informations d'état 374
    QTextBrowser 376
    Qu'est-ce que c'est ? 374
Algorithme
    générique 276
        qBinaryFind 277
        qCopy 277
        QCopyBackward() 285
        qDeleteAll 278
        qEqual() 285
        qFill 277
        qFind 276
        qSort 277
        qStableSort 278
        qSwap 278
AlignHCenter 57
alignment 137
Anticrénelage 186, 189
    basé sur X Render 197
AnyKeyPressed (QAbstract-ItemView) 231
API natives 444
append() 265
    QString 279
Apple Developer Connection 479

Application
Age 7
capable de gérer les plug-in 435
Carnet d'adresse 458
CD Collection 317
Cities 246
Color Names 240
Coordinate Setter 232
Currencies 243
Directory Viewer 238
écrire des plug-in 439
Flowchart Symbol Picker 230
Image Converter 303, 353
Image Pro 419
implémenter la fonctionnalité 77
infobulle et information d'état 374
Mail Client 154
MDI Editor 160
ouverte aux traductions 390
Project Chooser 216
Quit 7
Regexp Parser 251
Settings Viewer 234
Splitter 152
Spreadsheet 46, 92
Team Leaders 236
Tetrahedron 207-208
Threads 409
Tic-Tac-Toe 462
traduire 402
Trip Planner 343
Windows Media Player 448
apply() 422-423
applyEffect() 438-440
appTranslator 398, 401
Architecture modèle/vue 228
arg() 279, 392
QString 62
ARGB32 197
arguments() 330
QApplication 168
Assistant Qt 10, 379
asVariant() 452
at() 271
Attribut
WA_DeleteOnClose 74, 162
WA_StaticContents 117
Auto-Recalculate 52, 71, 75, 79
avg() 104

B

BackgroundColorRole 242, 256
Barre d'état, configurer 56
Barre d'outils
ancrable 157
créer 51
Base de données 309
connexion et exécution de requêtes 310
formulaires 321
présenter les données 317
beforeInsert() 320, 325
begin() 269
beginGroup() 71
Bibliothèque
C++ standard 523
dynamique 425, 488
E/S C++ 524
OpenGL 207
statique 488
STL 63
bindValue() 313
Boîte de dialogue
About 69
conception rapide 25
créer 15
de feedback 44
dynamique 39
étapes de création 26
extensible 33
Find 16
Go-to-Cell 26
intégrée 40
multiforme 32
multipage 39
QDialog (classe de base) 16
Sort 33
utiliser 64
bool 489
Boucle
d'événement 4, 178
foreach 271
while 417
boundingRect() (QPainter) 204
break 273
BufferSize 416
buttons() (QMouseEvent) 117

C

C#
différences avec C++ 489
différences avec Java 483
C++
chaînes de caractères 505
définitions de classe 490
destructeur 496
différences avec Java et C# 489
énumérations 507
espaces de noms 518
héritage 493
pointeurs 497
polymorphisme 493
préprocesseur 520
présentation 483
surcharge d'opérateur 512
tableaux 502
typeDef 509
types valeur 514
cachedValue 97, 100
cacheIsDirty 97, 98
cacheIsValid 100
canRead() 428, 429

canReadLine() 352
capabilities() 427
    CanRead 428
    CanReadIncremental 428
    CanWrite 428
Capuchon 185
cascade() 164
CaseSensitivity 17
cd() 333
cdModel 325
cdTableView 325, 327
Cell 79, 81, 90, 96
    arbre d'héritage 79
cell() 83, 90
cerr 486
Chaînes de caractères 505
changeEvent() 401
char 386, 489
characters() 362-363
Chargement 85
charset 224
checkable 33
Classe
    à accès aléatoire
        QBuffer 288
        QFile 288
        QTemporaryFile 288
    conteneur 263
        itérateurs 267
        QBrush 271
        QByteArray 266, 271
        QCache 275
        QDateTime 266
        QFont 271
        QHash 264, 273
        QImage 271
        QLinkedList 264-266
        QList 264
        QList<T> 266
        QMap 264, 273
        QPair 285
        QPixmap 271
        QRegExp 266
        QSet 275
        QString 266, 271
        QVariant 266
        QVarLengthArray <T,Prealloc> 285
        QVector 264, 266
    d'affichage d'éléments 227
        QListWidget 228
        QTableWidget 228
        QTreeWidget 228
        utiliser 229
    d'éléments 82
    de modèle 244
        QAbstractItemModel 244
        QAbstractListModel 244
        QAbstractTableModel 244
    de widgets 40
    QByteArray 278
    QString 278
    QVariant 278
    séquentielle
        QProcess 288
        QTcpSocket 288
        QUdpSocket 288
    TextArtDialog 437
    wrapper de plug-in 426
Clé 71
clear() 81, 82, 252, 278, 400
    Spreadsheet 58
clearCurve() 129
clicked() 7, 170, 304
Client
    FTP 330
    HTTP 339
    TCP 342
ClientSocket 349-350
clipboard() 88, 224
clone() 98, 514
close() 19, 61, 74, 164, 333
closeActiveWindow() 164
closeAllWindows() 74, 164
closeConnection(), TCP 348
closeEditor() 260
closeEvent() 70, 164, 167
    MainWindow 74
    QWidget 47, 61
codecForName() 389
CodeEditor 171
ColorGroup 115
ColorNamesDialog 240
column() (QModelIndex) 242
columnCount() 82, 244
columnSpan 148
COM, objets 448
commit() 313
commitAndCloseEditor() 259
commitData() 464
Communication inter-processus 303
Compagnon 18
compare() 68, 94-95
Compilateur
    exécution 486
    macros 446
    MinGW C++ 478
    moc 22
    uic 29
    unités 484
Composition (mode) 188, 198
CompositionMode_SourceOver 198
CompositionMode_Xor 199
connect() 19, 192, 419, 423
    QObject 9, 23
connected() 344, 345
connectionClosedByServer() 349
connectToHost() 333, 345
connectToServer() 345
Connexion
    aux bases de données 310
    de requêtes 310
    établir 6
    signal/slot 304
const 499
const_cast<T>() 510
constData() 283
Constructeur de copie 514
contains() (QRect) 117

Conteneur
associatif 264, 273
QCache 275
QHash<K,T> 264, 273
QMap<K,T> 264, 273
QSet 275
copie 270
séquentiel 264
QLinkedList 266
QLinkedList<T> 264-265
QList<T> 264, 266
QQueue<T> 266
QStack<T> 266
QVector 264
QVector<T> 264, 266
contentsChanged() 166-167
contextMenuEvent() (QWidget) 55, 67
ContiguousSelection 81, 89
continue 273
Contrôle ActiveX 448
controllingUnknown() 456
Conversions de type 509
ConvertDepthTransaction 422
convertTo1Bit() 420
convertTo32Bit() 420
convertTo8Bit() 420
convertToFormat() 112
Coordonnées
conversion logiques-physiques 190
système par défaut 188
copy() 88
CopyAction 218
copyAvailable() 162
count() 265
Courbe de Bézier 186
cout 486
Crayon 185
create() 121, 427-428
createActions() 49, 376, 398
createConnection() 311, 313
createContextMenu() 49
createEditor() 161, 258, 260
createIndex() 249
createLanguageMenu() 399-400
createMenus() 49, 164, 398
createStatusBar() 49, 56
createToolBars() 49
critical() (QMessageBox) 59
curFile 62, 167
currencyAt() 246
CurrencyModel 243
currentCdChanged() 324
currentCellChanged() 57
currentDateTime() (QDateTime) 193
currentFormula() 84
currentIndex() (QComboBox) 68
currentItem 364
currentLocation() 84
currentRow 151
currentRowChanged() 151
currentText 363
currentThread() 417
Curseur 452
CursorHandler 428, 434
curveMap 129
curZoom 127
CustomerInfoDialog 176
cut() 88
Editor 163

D

data() 99, 244-245, 283
QMimeData 219
QTableWidgetItem 96
dataChanged() 249
QClipboard 225
Datagramme UDP 353
DataSize 416
debug 5
Déclaration préalable 16
default 33
defined() 522
Définition de classe 490
Dégradé
circulaire 188
conique 188
linéaire 187
del() 90
Délégué 228
personnalisé 256
delete 90, 252
deleteLater() 350
delta() 135
Dérivation
QDialog 16
QMainWindow 46
QMimeData 224
QTableWidget 78
QTableWidgetItem 96
QWidget 108
Dessin
courbe de Bézier 186
crayon 185
dégradés
circulaires 188
coniques 188
linéaires 187
formes géométriques 186
pinceau 185
police 185
QPainter 184
rectangle sans anticrénelage 189
styles
de capuchon et jointure 185
de crayon 185
tracé 187
viewport 189
DiagCrossPattern 187
Dinkumware 527
Directives 445
#include 486
de préprocesseur 487
Disabled (groupe de couleurs) 115
disconnect() 423

disconnected() 344, 349
DisplayRole 97, 99, 242, 245
Disposition 7, 28, 144
  empilée 150
  gérer 143
  gestionnaires 9, 146
  manuelle 145
  positionnement absolu 144
distances 248
DLL 425, 488
Document
  multipage 201
  multiple 72
Documentation de référence 10
documentElement() 367
documentTitle() 378
documentWasModified() 166
DOM (Document Object Model) 359
DomParser 367
done() 231, 331
Donnée
  binaire
    écriture 288
    lecture 288
  les stocker en tant qu'éléments 82
  présentation sous forme tabulaire 317
  types primitifs 489
DontConfirmOverwrite (QFileDialog) 61
double 489
  mise en mémoire tampon 122
dragEnterEvent() 214-215
dragMoveEvent() 218
draw() 184, 194, 209
  dessin d'un tétraèdre 210
drawCurves() 136, 138
drawDisplay() 259
drawFocus() 259
drawGrid() 136
drawLine() 196
  QPainter 114
drawPie() 186
drawPolygon() 195
drawPolyline() 138
drawPrimitive() 130
  QStylePainter 130
drawRect() 189
drawText() 137, 196
  conversion des coordonnées 190
dropEvent() 214
  glisser-déposer 215
dumpdoc 451
duration() 193
dynamic_cast<T>() 510
dynamicCall() 452

E

E_NOINTERFACE 456
Echelon 140
Ecriture
  code XML 370
  données binaires 288
  texte 294
Editeur
  de connexion (Qt Designer) 37
  de liens 485
editingFinished() 259
Editor 164
  cut() 163
  fonctions 165
  save() 162
EditRole 97-99, 242
effects() 436, 439
Ellipse (dessiner) 186
emit 21
enabled 27
enableFindButton() 19, 21
end() 269
  conteneur non const 271
endElement() 360, 362, 364
endGroup() 71
endl 486
endsWith() 281
enterErrorState() 431
Entrées/sorties 287
  lire et écrire des données binaires 288
  lire et écrire du texte 294
  parcourir les répertoires 300
entryHeight() 203, 204
entryList() 301
Enumérations 507
Equivalence pointeurs/tableaux 503
eraseRect() 440
error() 306, 344, 349
errorString() 363
escape() 221
Espace de noms 518
  std 125
Espacement 20
evalExpression() 100-101, 104
evalFactor() 101, 104
evalTerm() 101-104
Evénement 4
  boucle 178
  close 164
  drop 214
  filtre 175, 177
  gestionnaires, réimplémenter 170
  key 170
  paint 113, 118
  propagation 178
  resize 131
  timer 172
  traiter 169, 177
  versus signal 170
  wheel 135
event() 170
eventFilter() 176
exec() 66, 312, 418
  QApplication 178
  QDialog 66
  QPrintDialog 200
  requête SQL 311

Exécutable 485
intégrer des données 302
execute() 307
QProcess 307
exitAction 53
expand() 239
Expanding 149
explicit 511
Expression 101
régulière 250
syntaxe 101
extern 517

F

faceAtPosition() 211-212
fatalError() 362, 364
Fenêtre
d'application
QDialog 4
QMainWindow 4
dessin 188
modale 66
non modale 64
principale
configurer 49
créer 45
Fichier d'en-tête 18, 48
<QtAlgorithms> 276
<QtGui> 18
définition 487
Fichier objet 485
fill() (QPixmap) 136
fillRect() (QPainter) 116
Filtre
cin en cout 299
d'événement 446
installer 169, 175
entrées du système
de fichiers 238
find() 444
findChild<T>() (QObject) 40
findClicked() 19, 21
findNext() 17, 21, 65, 91
findPrevious() 17, 21, 65, 92
finished() 306, 422
firstChild() 369
flags() 247, 249
flipHorizontally() 420
FlipTransaction 422
flipVertically() 420
float 489
flush() 289
focusNextChild() 175, 177
Fonction
globale 485, 516
prototype 486
virtuelle pure 494
fontMetrics() (QWidget) 174
FontRole 242, 256
foreach 75, 505
foreground() (QPalette) 115
FOREIGN KEY 318
forever 347
formats() 222-223
ARGB32 prémultiplié 197
formula() 83
Formulaire
créer à l'aide de QWidget 108
développement 26
disposition des widgets 28
insérer un HexSpinBox 118
maître/détail 321
modifier la conception 32
nommer les widgets 35
Frameworks de migration Qt/Motif et Qt/MFC 444
Froglogic 444
fromAscii() 283
fromLatin1() 283
fromPage() (QPrinter) 205
ftpDone() 331
FtpGet 330
ftpDone() 331
ftpListInfo(), urlInfo 336

G

generateDocumentation() 451
generateId() 320
generateRandomTrip() 351
geometry() (QWidget) 72
Gestion de session
tests et débogage 467
X11 461
Gestionnaire
d'événements
endElement() 360
réimplémenter 170
startElement() 360
de disposition 9, 146
QGridLayout 9, 146
QHBoxLayout 9, 146
QVBoxLayout 9, 146
get() 330, 332-333, 337
opérations HTTP 339
getColor() (QColorDialog) 211
getDC() 445
getDirectory() 335
getFile() 330, 332
opérations HTTP 340
GetInterfaceSafetyOptions() 456
getOpenFileName() (QFileDialog) 59
getSaveFileName() (QFileDialog) 61
GL_SELECT 212
glClearColor() 209
glClearIndex() 209
glColor3d() 210
glIndex() 210
Glisser, types personnalisés 219
Glisser-déposer 213
activer 214
QTableWidget 219
GNU (General Public License) 478
Go to Cell 27
GoToCellDialog 27

Graphique
2D 183
3D 183
OpenGL 207
group() 120
Groupe de couleurs
Active 115
Disabled 115
Inactive 115
guidgen 457
GuiServer 471

H

Handle 444, 446
hasAcceptableInput() (QLineEdit) 32
hasFeature() 313
hasLocalData() 418
hasNext() 267
hasPendingEvents() 181
hasPrevious() 268
head() 341
headerData() 244-245, 248, 256
Hello Qt 4
HelpBrowser 377, 380
Héritage 493
HexSpinBox 106, 108, 118
hide() 128
hideEvent() 173, 175
hostName 392

I

IANA (Internet Assigned Numbers Authority) 215
icon() 120
IconEditor 109, 115, 120, 122
pixelRect() 115
IconEditorPlugin 120
iconForSymbol() 231
iconImage() 109-110, 113
Identifiant de la fenêtre 444
IgnoreAction 218
image() 110, 421
presse-papiers 224
imageCount() 433
imageSpace() 300-301
ImageWindow 420
Impression 199
QPainter 205
QTextDocument 202
Inactive (groupe de couleurs) 115
includeFile() 120
incomingConnection() 349-350
index() 253
de modèle 249
indexOf() 150, 280
Infobulle 374
information() (QMessageBox) 59
initFrom() 130, 136, 198
initializeGL() 208-209
insert() 237, 281
avec itérateur 269
dans map 273
insertMulti() 274-275
insertRow() 233, 315
installEventFilter() 176-177
instance() (QPluginLoader) 438
Instruction
DELETE 314
GET 337
if 417
INSERT 313-314
SELECT 312, 314
UPDATE 314
int 312, 489
Interface avec les API natives 444
Intermediate 107
QSpinBox 107
Internationalisation 385
passer dynamiquement d'une langue à l'autre 396
traduire les applications 390, 402
Unicode 386
Internet Explorer
accès à l'aide 380
intégrer un widget 453
options de sécurité du composant 455
Introspection 25
Invalid 107
QSpinBox 107
invokeMethod() 424
IObjectSafety 455
isActive() 312
isalpha() 387
isContainer() 121
isdigit() 387
isEmpty() 282
conteneur 269
isLetter() 387
isLetterOrNumber() 387
isLower() 387
isMark() 387
isNumber() 387
isPrint() 387
isPunct() 387
isSessionRestored() 467
isSpace() 387
isSymbol() 387
isUntitled 165, 167
isUpper() 387
item->text() 231
itemChanged() 81
ItemIsEditable 249
Itérateur
de style Java 267
de style STL 269
mutable 268
pour liste chaînée 265

J

Java
différences avec C# 483
différences avec C++ 489
machines virtuelles 470

join() 282
Jointure 185
JournalView 401
jumpToNextImage() 433

K

KDE XIV
  fermeture de session 461
  projet XVIII
key() 138, 169, 246, 275-276
  map 275
keyPressEvent() 135, 170-171, 175
  Plotter 139
keyReleaseEvent() 170
keys() 274, 427
  itérateur STL 276
killTimer() (QObject) 175
Klarälvdalens Datakonsult 444

L

LanguageChange 402
LargeGap 203
Latin-1 386
LD_LIBRARY_PATH 481
leaders() 238
Lecture
  code XML 360, 365
  données binaires 288
  texte 294
left() 280, 283
LeftDockWidgetArea 158
length() 279, 282
LinkAction 218
list() 333
Liste chaînée 265
load() 394, 401
loadFile() 59, 64
loadPlugins() 437-438
localeAwareCompare() 281, 395
LocaleChange 401-402
localeconv() 395
lock() 411
login() 332, 333
long 489
long long 489
lrelease 402
lupdate 402, 403

M

Mac OS X
  identifier la version 446
  installer Qt 479
macEvent() 448
macEventFilter() 448
Macro
  de compilateur 446
  Q_ OBJECT 47
  Q_DECLARE_INTERFACE () 436
  Q_DECLARE_METATYPE() 284
  Q_EXPORT_PLUGIN2() 121, 428, 441
  Q_INTERFACES() 120, 439
  Q_OBJECT 17-18, 21, 110, 390, 439
  Q_PROPERTY() 109
  qPrintable() 283, 289
  QT_TR_NOOP() 392
  QT_TRANSLATE_NOOP() 393
  SIGNAL() 7, 23
  signals 17
  SLOT() 7, 23
  slots 17
MagicNumber 86, 292
mainSplitter 154
MainWindow 399
  changeEvent() 401
  closeEvent() 74
  dérivation de QMainWindow 47
  glisser-déposer 214
  newFile() 82
  sort() 69
  switchLanguage() 401
  updateStatusBar() 84
make 5, 479
Makefile 5, 22, 29
Manipulateur de flux 295
Masque
  AND 431
  XOR 431
Matrice world 190
  dessin 188
MDI (Multiple Document Interface) 75, 159
Mécanisme
  des ressources 50
  fenêtre-viewport 189
  parent-enfant 31
MediumGap 204
Mémoire d'image 470
Menu
  Bar() (QMainWindow) 53
  créer 51
  Edit 88
  File 57
  Options 92
  Tools 92
  Window 160
message() 423
metaObject(), déclaration avec Q_OBJECT 25
Méta-objets 25
mid() 280, 283
  QString 66
Migration de Motif/Xt et MFC vers Qt 444
MIME (type) 215
mimeData() 224
MinGW C++ 478
minimumSizeHint() 124, 129, 149

mirrored() 423
Mise en veille 462
mkdir() 239, 333
moc 22, 25
Mode de composition 188, 198
Modèle
  arborescence 242
  index 249
  liste 242
  personnalisé 241
  prédéfini 236
  tableau 242
modified() 80, 85
modifiers() (QKeyEvent) 135
Module
  QtCore 18
  QtGui 18
  QtNetwork 18
  QtOpenGL 18, 207
  QtSql 18
  QtSvg 18
  QtXml 18
mouse 169, 208
MouseButtonPress 170
mouseMoveEvent() 117, 136, 210
mousePressEvent() 117, 136, 170, 192, 210, 465
  QListWidget 217
  QWidget 116
mouseReleaseEvent() 136, 139
  collage avec le bouton du milieu de la souris 225
move() (QWidget) 72
MoveAction 218
moveToThread() 419
Multithread 407
mutable 97, 101
Mutex 411
  emploi 413
MVC (Modèle-Vue-Contrôleur) 228

N

name() 120
New 51
newFile() 51, 58, 161, 166
  MainWindow 73, 82
newPage() 199, 200, 205
next() 267, 268
  map 275
nextBlockSize() 344-345, 347, 350
nextSibling() 369
nmake 5
NoBrush 187
Node 250, 254
nodeFromIndex() 253-254
Nœud 251
normalized() (QRect) 130, 133
notify() (QApplication) 178
number() (QString) 107
numCopies() (QPrinter) 205
numRowsAffected() 312
numTicks 140
numXTicks 125
numYTicks 125

O

objectName 27
Objet
  COM 448
  compare 68, 94
  QSettings 72
  SpreadsheetCompare 68
offset 173, 174
offsetOf() 250
okToContinue() 58, 165, 167
on_browseButton_clicked() 304
on_convertButton_clicked() 304
on_lineEdit_textChanged() 32
Open 52
Open source 478
open() 59, 162, 166
  QTemporaryFile 307
OpenGL 183, 207
  bibliothèque 207
openRecentFile() 52, 64
Opérateur
  () 94
  + 279
  += 279
  . (point) 493
  : 486, 495
  << 486
  -> (flèche) 497
  bit à bit | 508
  bit à bit|= 508
  d'affectation 514
  sizeof() 502
  unaire & 497
  unaire * 499
operator 291
operator()(int) 95
operator==() 274
operator>>() 291
Ordre de tabulation 22, 28
OvenTimer 191

P

Page d'accueil 75
paginate() 202-203
paint 113
paintEngine() 445
Painter, transformations 188
paintEvent()
  anticrénelage 197
  copie de pixmap 136
  définition du viewport 194
  IconEditor 113
  implémentation de event() 170
  OvenTimer 192
  réimplémentation TicTacToe 465
  widget Ticker 174
paintGL() 208-209

palette() (QWidget) 115
Paramètre
charset 224
parent 244, 247
preferredType 223
section 245
stocker 70
parent 244, 247
parent() (QModelIndex) 242
parse() 365
parseEntry() 368
Partage implicite 270
principe 272
paste() 90
PATH 5, 380
PdfFormat 199
peek() 294, 429
penColor() 109, 111, 113
pendingDatagramSize() 357
Pile 498
Pile de zoom
curZoom 127
zoomStack 127
Pilote
QODBC 310
QPSQL 310
QSQLITE 310
QSQLITE2 310
QTDS 310
Pinceau 185
de fond 188
origine 188
pixelRect() 116
IconEditor 115
pixmap() 224, 440
Plastique 12
PlotSettings 123, 125, 127, 139
adjust() 134
Plotter 122, 124
keyPressEvent() 139
Plug-in 119
création 425
d'application 435, 439
développement de Qt 426
plugins 122
Pointeur
à réticule 132
C++ 497
conteneur 252
d'attente standard 86
de fichier 288
de nœud 256
glisser-déposer 218
nul 17
QObject 64
QPointer 498
toupie 7
hexadécimal 106
vers la barre d'état 56
void 83
Police 185
Polymorphisme 493
pop() 266
populateListWidget() 437-438
pos 100
post() 339, 341
postEvent() 423
Preferred 149
preferredType 223
prepare() 312
prepend() 62
Préprocesseur 488, 520
Presse-papiers
de sélection 224
gérer 224
previous() 268
map 275
printBox() 206
printFlowerGuide() 202
printHtml() 201
printImage() 200
printPages() 202, 205
PrintWindow 200
pro (fichier) 122
processEvents() (QApplication) 179
processNextDirectory() 335, 337
processPendingDatagrams() 356
Programmation embarquée 469
ProjectListWidget 216, 218
Promotion 118
propertyChanged() 455
Protocole
de transport
TCP 329
UDP 329
QCOP 471
Prototype de fonction 486
push() 266
put() 330, 333

Q

Q_ OBJECT 47
Q_CLASSINFO() 459
Q_DECLARE_INTERFACE() 436
Q_ENUMS() 449
Q_EXPORT_PLUGIN2() 121, 428
Q_INTERFACES() 120, 439
Q_OBJECT 17-18, 21, 110, 390, 439
Q_OS_UNIX 446
Q_OS_WIN 446
Q_PROPERTY() 109
Q_WS_ X11 446
Q_WS_MAC 446
Q_WS_QWS 446
Q_WS_WIN 446
qAbs() 278
QAbstractItemDelegate 258
QAbstractItemModel 244, 252
reset() 246, 250
QAbstractItemView 52, 156, 218
AnyKeyPressed 231
ContiguousSelection 81, 89
NoEditTriggers 233
selectAll() 91
setEditTriggers() 231
QAbstractListModel 244

QAbstractScrollArea 42, 82, 156
QAbstractSocket 342
    waitForDisconnected() 424
QAbstractTableModel 244
QAction 55, 162, 165, 171-172, 374, 399, 400
    conversion d'un pointeur QObject 64
    setShortcutContext() 172
QActionGroup 52, 162-163, 400
    triggered() 400
qApp 53
QApplication 4, 394, 418
    arguments() 168
    clipboard() 88, 224
    commitData() 462
    exec() 178
    filtre d'événement 177
    GuiServer 471
    isSessionRestored() 467
    notify() 178
    processEvents() 179
    quit() 61
    setLayoutDirection() 394
    setOverrideCursor() 132
    translate() 392
QAssistantClient 379
    showPage() 380
QAXAGG_IUNKNOWN 456
QAxAggregated 456
QAxBase 449
    dynamicCall() 452
    generateDocumentation() 451
    queryInterface() 452
QAxBindable 453
QAXCLASS() 459
QAxContainer 448
    schéma d'héritage 450
QAXFACTORY_BEGIN() 459
QAXFACTORY_DEFAULT() 454, 457
QAXFACTORY_END() 459
QAXFACTORY_EXPORT() 459
QAxObject 449
QAxServer 448, 453
qaxserver.def 457
qaxserver.rc 457
QAXTYPE() 459
QAxWidget 449
    setControl() 451
qBinaryFind() 264, 277
QBitArray 431
QBrush 116, 186
QBuffer 86, 288
    fichiers à télécharger 339
QByteArray 218-219, 222, 266, 278, 288, 346, 352
QCDEStyle 130
QChar 386
    isDigit() 387
    isLetter() 387
    isLetterOrNumber() 387
    isLower() 387
    isMark() 387
    isNumber() 387
    isPrint() 387
    isPunct() 387
    isSpace() 387
    isSymbol() 387
    isUpper() 387
QCheckBox 40, 147, 178
QClipboard
    dataChanged() 225
    Selection 224
    setText() 88
QCloseEvent, accept() 411
QColor 111, 116
QColorDialog 208
    getColor() 211
QComboBox 158
    addItem() 241
    currentIndex() 68
QComboBoxes 176
qCompress() 294
qconfig 472
QCOP 471
QCopChannel 471
qCopy() 277
QCoreApplication 330, 418
    addLibraryPath() 435
    arguments() 330
    postEvent() 423
    removePostedEvent() 423
QDataStream 85-86, 288, 290, 351, 431, 490
    numéro de version 292
    opérations TCP 342
    Qt_4_1 86
    skipRawData() 431
QDate (toString) 395
QDateEdit 395
QDateTime 266
    currentDateTime() 193
    toString 395
QDateTimeEdit 395
qDeleteAll() 252, 278
QDesignerCustomWidget-CollectionInterface 122
QDesignerCustomWidget-Interface 119
    IconEditorPlugin 120
QDialog 4, 16, 31, 178
    dériver 16
    exec() 66
    Rejected 66
QDir 300, 301
    entryInfoList() 301
    entyList() 301
    exists() 301
    imageSpace() 301
    mkdir() 301
    rename() 301
    rmdir() 301
QDirectPainter 471
QDirModel 236, 238, 240
    mkdir() 239
QDockWidget 157
    setFeatures() 157
QDockWindow 157

QDomDocument
documentElement() 367
save() 370
setContent() 367
QDomElement 369
QDomNode 368
firstChild() 369
nextSibling() 369
tagName() 368
toElement 368
QDoubleValidator 31
QDrag 217
start() 218
QDragEnterEvent 218
QEvent 170
MouseButtonPress 170
type() 170
QFile 85-86, 288, 301, 336
exists() 301
remove() 301
QFileDialog
DontConfirmOverwrite 61
getOpenFileName() 59
getSaveFileName() 61
QFileInfo 301
qFill() 277
qFind() 276
QFont 186, 288, 292
QFontMetrics 174
size() 174
QFrame 41, 121
QFtp 329
cd() 333
close() 333
commandFinished() 333
commandStarted() 333
ConnectToHost 333
done() 332
ftpListInfo() 335
get() 330, 333
list() 333
listInfo() 335
login() 333
mkdir() 333
put() 330, 333
rawCommand() 333
readyRead() 339
remove() 333
rename() 333
rmdir() 333
stateChanged() 333
qglClearColor() 209
qglColor() 210
QGLWidget
dessin avec OpenGL 207
qglClearColor() 209
setFormat() 209
QGridLayout 9, 19, 143, 146-147
QGroupBox 178
QHash 274
QHash() 275
QHBoxLayout 9, 19, 143, 146-147
QHeaderView 82
QHostInfo
fromName() 355
lookupHost() 355
QHttp 329
done() 340, 342
get() 339, 341
getFile() 340
head() 341
httpDone() 340
post() 339, 341
QTcpSocket 341
QtSslSocket 341
read() 342
readAll() 342
readyRead() 342
request() 341
requestFinished() 342
requestStarted() 342
setHost() 341
setUser() 341
QIcon 231
QImage 111, 288, 427
affichage de haute qualité 197
CompositionMode_Source-Over 198
CompositionMode_Xor 199
format ARGB32 prémultiplié 197
imprimer 199
mirrored() 423
rect() 117
setPixel() 117
QImageIOHandler 428
QImageIOPlugin 427-428
QImageReader 427-428
QInputDialog 42
QIntValidator 31
QIODevice 287, 293
écrire dedans 300
opérations HTTP 341
peek() 294, 429
QBuffer 287
QFile 287
QProcess 287
QTcpSocket 287
QTemporaryFile 287
QUdpSocket 287
ReadOnly 87
seek() 294
unget-Char() 294
WriteOnly 87
QItemDelegate 258
drawDisplay() 259
drawFocus() 259
QItemSelectionModel 324
QKeyEvent 171, 177
modifiers() 135
QLabel 42
boîte de dialogue Find File 147
fenêtre d'application 4
Hello Qt 4
implémentation 108
indicateurs d'état 56

setText() 424
statut de la dernière opération 343
QLatin1String() 393
QLayout 146
SetFixedSize 38
setMargin() 147
setSpacing() 147
QLibrary 425
QLineEdit
activer avec barre d'espace 175
boîte de dialogue Find File 147
entrées de données 42
hasAcceptableInput() 32
stocker une chaîne de filtre 240
text() 66
QLineEdits 355
QList 274
QListView 229, 237, 314
QListWidget 39, 151, 228, 231, 435
glisser-déposer 215
QListWidgetItem 231, 439
QLocale 395
System() 394
qlonglong 490
QMacStyle 130
QMainWindow 4, 78, 144, 157
dériver 46
menuBar() 53
statusBar() 56
qmake 5, 21-22, 29, 479
QMatrix 191
qMax() 278
QMenu 53
addAction() 54
addMenu() 54
QMenuBar 4
QMessageBox
about() 70
critical() 59
Default 58
Escape 58
information() 59
question() 59
warning() 58, 70
QMetaObject 25
invokeMethod() 424
QMimeData 213, 217
conversion en TableMime-Data 224
data() 219
dériver 224
glisser-déposer 219
setData() 219
text() 219
urls() 215
qMin() 194, 278
QModelIndex 237, 240, 245, 253
column() 242
parent() 242
row() 242
QMotifStyle 130
QMouseEvent 170
buttons() 117
QMovie 427
qmPath 398
QMultiMap 274
QMutex 407, 409, 411, 416, 423
QMutexLocker 412, 423
QObject 24, 266, 424, 439
connect() 9, 419, 423
deleteLater() 424
disconnect() 423
event() 170
filtre d'événement 177
findChild() 40
IconEditorPlugin 120
killTimer() 175
mécanisme parent-enfant 31
moveToThread() 419
QProcess 424
QTimer 424
sender() 64
setProperty() 451
startTimer() 174
tr() 18, 390
qobject_cast() 64, 218, 224, 510
QPaintDevice 445
QPainter 114, 136, 183
boundingRect() 204
dessiner 184
sur QPrinter 200
draw...() 184
drawLine() 114
fillRect() 116
imprimer 205
programmation embarquée 471
rotate() 191
scale() 191
setCompositionMode() 198
shear() 191
système de coordonnées 188
translate() 191
QPainterPath 186
QPalette 115
ColorGroup 115
foreground() 115
QPen 186
QPixmap 124, 288, 439
fill() 136
QPlastiqueStyle 130
QPluginLoader 438
QPoint 114
QPointer 498
QPointF 138
qPrintable() 289
QPrintDialog 200
choix de l'imprimante 199
options d'impression 205
setEnabledOptions() 205
QPrinter 200
fromPage() et toPage() 205
newPage() 199
numCopies() 205
setPrintProgram() 199

QProcess 86, 288, 303, 424
  execute() 307
  start() 305
  waitForFinished() 424
QProgressBar 42, 343
QProgressDialog 42, 179-180
QPushButton 40
  clicked() 7
  effets 3D 187
  implémentation 108
  répondre aux actions utilisateur 6
  setText() 39
  show() 11
  signal clicked() 170
QRadioButton 40
QReadLocker 413, 423
QReadWriteLock 411, 413, 423
QRect 115-117, 132
  contains() 117
  normalized() 130, 133
QRegExp 104, 266
  PatternSyntax 241
QRegExpValidator 31, 67, 107
qRegisterMetaTypeStream-Operators<T>() 285
qRgb() 111
qRgba() 111
QRubberBand 136
QScrollArea 135, 144, 155-156
  viewport() 155
  widgets constitutifs 155
QScrollBar 82
QSemaphore 407, 411, 413, 423
QSessionManager 463
QSetting 291
QSettings 71-72, 234
QShortcut 172
  setContext() 172
QSizePolicy 149
  Expanding 126, 149
  Fixed 149
  Ignored 149
  Maximum 149
  Minimum 112, 149
  MinimumExpanding 149
  Preferred 126, 149
QSlider 7, 8
qSort() 264, 277
QSortFilterProxyModel 236, 240
QSpinBox 7, 8, 106, 118, 158
  Acceptable 107
  Intermediate 107
  Invalid 107
  textFromValue() 107
  valueFromText() 107
QSpinBoxes 176
QSplashScreen 75
QSplitter 78, 144, 152, 159
  setSizes() 154
  sizes() 269
QSqlDatabase 310-311, 313
QSqlDriver 313
QSqlQuery 310-312, 314
  first() 312
  last() 312
  previous() 312
  seek() 312
QSqlQueryModel 236
QSqlRelationalDelegate 323
QSqlRelationalTableModel 236, 310
  formulaires 322
QSqlTableModel 236, 310, 314-315, 319
  QSqlRelationalTableModel 317
qStableSort() 94-95, 278
QStackedLayout 150
QStackedWidget 39, 150-151
QStackLayout 143
QStatusBar 4
  addWidget() 57
QString
  append() 279
  arg() 62, 392
  chaîne
    de caractères 507
    Unicode 387
  comparaison 277
  concaténation 279
  emploi comme conteneur 278
  localeAwareCompare() 395
  menu Edit 88
  mid() 66
  number() 107
  partage implicite 272
  replace() 221
  split() 90, 282
  sprintf() 279
  toInt() 66, 107
  toUpper() 107
  type de valeur 266
QStringList 94, 168, 201
  arguments de ligne de commande 330
  concaténation 282
  fichiers à télécharger 338
  recentFiles 63
  takeFirst() 297
QStringListModel 237, 240
QStyle 130
  PE_FrameFocusRect 130
QStyleOptionFocusRect 130
QStylePainter 130
qSwap() 250, 278
Qt
  AlignHCenter 57
  BackgroundColorRole 242, 256
  CaseInsensitive 17
  CaseSensitive 17
  classe d'éléments 82
  DiagCrossPattern 187
  DisplayRole 231, 242
  EditRole 231, 242
  FontRole 242, 256
  IconRole 231
  ItemIsEditable 249

LeftDockWidgetArea 158
mécanisme des ressources 50
NoBrush 187
SolidPattern 187
StatusTipRole 242
StrongFocus 126
système de méta-objets 25
TextAlignmentRole 242, 256
TextColorRole 242, 256
ToolTipRole 242
UserRole 231
WA_DeleteOnClose 166
WA_GroupLeader 378
WA_StaticContents 110
WaitCursor 132
WhatsThisRole 242
Qt Designer 26
créer
des widgets personnalisés 108
un formulaire 151
éditeur de connexions 37
intégrer des widgets personnalisés 118
Qt Linguist
exécution 403
présentation 403
Qt Quaterly 12
Qt/Embedded 469
Qt/Mac (installer) 479
Qt/Windows (installer) 478
Qt/X11(installer) 479
qt_metacall(), déclaration avec Q_OBJECT 25
QT_TR_NOOP() 392
QT_TRANSLATE_NOOP() 393
QTableView 93, 229, 314, 320
QTableWidget 78, 228, 232, 401
ajouter le glisser-déposer 219
attributs QTableWidgetItem 79
dériver 78
implémentation 108
opérations TCP 343
QHeaderView 82
QScrollBar 82
selectColumn() 91
selectedRanges() 89
selectRow() 91
setCurrentCell() 66
setItem() 84, 232
setItemPrototype() 98
sous classe Spreadsheet 49
Track Editor 257
widgets constitutifs 82
QTableWidgetItem 79, 82, 90, 222, 232, 235
data() 96, 99
dériver 96
text() 96
QTableWidgetSelectionRange 222
QTabWidget 39, 41
QtCore 18, 299
QTcpServer 329, 342
QTcpSocket 86, 288, 329, 341, 342
canReadLine() 352
connected() 345, 346
connectionClosedByServer() 349
disconnected() 349
error() 349
listen() 352
readClient() 351
readLine() 352
readyRead() 347
seek() 346
updateTableWidget() 347
write() 346
QTDIR 122
QTemporaryFile 288, 307
QTextBrowser 42, 379
moteur d'aide 376
QTextCodec 388
codecForName() 389
setCodecForCStrings() 389
setCodecForTr() 389, 403
QTextDocument 202
QTextEdit 42, 78, 149, 156, 307
glisser-déposer 214
QTextStream 86, 288, 294, 299, 388
AlignAccountingStyle 296
AlignCenter 296
AlignLeft 296
AlignRight 296
FixedNotation 296
ForcePoint 296
ForceSign 296
internationalisation 388
opérations
TCP 342
XML 370
readAll() 295
readLine() 295
ScientificNotation 296
setRealNumberNotation() 296
SmartNotation 296
UppercaseBase 296
UppercaseDigits 296
QtGui 18
QThread 407
currentThread() 417
exec() 424
run() 408
terminate() 409
wait() 411
QThreadStorage 423
QTime (toString) 395
QTimeEdit 259, 323, 326
Track Editor 260
QTimer 175, 192, 424
sendDatagram() 354
QtNetwork 18
QToolBar 4
QToolBox 41
QToolButton 40, 127, 158

QtOpenGL 18, 207
Qtopia Core 469
  formats de police 473
  prise en charge de VNC 472
Qtopia PDA 470
Qtopia Phone 470
Qtopia Platform 470
QTranslator 398, 400, 404
  load() 394
QTreeView 229, 239, 256
QTreeWidget 39, 147, 149, 158, 228, 233-234, 367
  fichier d'index 361
  opérations XML 363, 365
QTreeWidgetItem 369, 460
  opérations XML 363
QtSql 18
QtSslSocket 341
QtSvg 18
qtTranslator 398, 401
QtXml 18
Qu'est-ce que c'est ? 375
QUdpSocket 86, 288, 329, 353-354
  writeDatagram() 355
queryInterface() 452, 456
querySubObject() 452
question() (QMessageBox) 59
QUiLaoder 39
quint32 291
quit() (QApplication) 61
quitOnLastWindowClosed 61
qUncompress() 294
QUrl 334
QUrlInfo 336
QVariant 63, 71, 82, 99, 223, 266, 278, 283, 288, 291, 312
  double 283
  int 283
  QBrush 283
  QColor 283, 284
  QCursor 283
  QDateTime 283
  QFont 283, 284
  QIcon 284
  QImage 284
  QKeySequence 283
  QPalette 283
  QPen 283
  QPixmap 283, 284
  QPoint 283
  QRect 283
  QRegion 283
  QSize 283
  QString 283
QVBoxLayout 9, 19, 143, 146, 147
QVector 124, 138
qvfb 471
QWaitCondition 407, 411, 416, 423
  wait() 417
QWhatThis, createAction() 376
QWidget 7, 78
  changeEvent() 401
  close() 19, 61
  closeEvent() 47, 61
  contextMenuEvent() 55, 67
  dériver 108
  event() 171
  find() 444
  fontMetrics() 174
  geometry() 72
  mousePressEvent() 116
  move() 72
  palette() 115
  QGLWidget 207
  repaint() 113
  resize() 72
  scroll() 175
  setCursor() 132
  setGeometry() 72
  setLayout() 9
  setMouseTracking() 117
  setStyle() 130
  setTabOrder() 22
  setToolTip() 374
  setWindowIcon() 49
  setWindowTitle() 8
  sizeHint() 20, 38, 57
  style() 130
  update() 112
  updateGeometry() 112
  windowModified 62
  winID() 444
QWindowsStyle 130
QWindowsXPStyle 130
QWorkspace 78, 144, 159, 167
QWriteLocker 413, 423
QWS_DEPTH 472
QWS_KEYBOARD 471
QWS_MOUSE_PROTO 471
QWS_SIZE 472
qwsEvent() 448
qwsEventFilter() 448
QWSServer 473
qwsServer (variable globale) 473
QXmlContentHandler 362, 365
  endElement() 360
  startElement() 360
QXmlDeclHandler 360
QXmlDefaultHandler 361-362
  errorString() 363
QXmlDTDHandler 360
QXmlEntityResolver 360
QXmlErrorHandler 360, 362, 365
QXmlInputSource 365
QXmlLexicalHandler 360
QXmlSimpleReader 360, 365

R

range.rightColumn() 68
Rational PurifyPlus 490
rawCommand() 333
Réactivité et traitement intensif 178
read() 433
  QHttp 342

readAll() 295
QHttp 342
QIODevice 293
readBitmap() 434
readDatagram() 357
readFile() 87
Spreadsheet 60
readHeaderIfNecessary() 430, 433
readLine() 295, 297, 352
ReadOnly (QIODevice) 87
readRawBytes() 290, 291
readSettings() 49, 70-71, 155, 234, 235
readyRead() 339, 342, 344
recalculate() 92, 93
recentFileActions 52
recentFiles 62, 63
record() 327
rect() (QImage) 117
Référence 500
refreshPixmap() 128, 131, 135
refreshTrackViewHeader() 324, 327
regExpChanged() 256
RegExpModel 250, 256
RegExpParser 250, 256
RegExpWindow 250
Registre système
enregistrement
d'un serveur ActiveX 460
du serveur 457
stockage des paramètres 71
regsvr32 457
reinterpret_cast() 511
reject() (QDialog) 31, 36
release() 5, 456
releaseDC() 445
remove() 281, 333
avec itérateur 268
removeAll() 62
removePostedEvents() 423
removeRows() 237, 316
rename() 333
repaint() (QWidget) 113
Répertoire (parcours) 300
replace() 281
QString 221
Représentation binaire des types 86
requestFinished() 342
requestPropertyChange() 455
requestStarted() 342
Réseau
envoi et réception de datagrammes UDP 353
gestion 329
programmer
les application client/serveur TCP 342
les clients FTP 330
les clients HTTP 339
reserve() 275
reset() (QAbstractItemModel) 246, 250
resize() (QWidget) 72, 127, 131
resizeEvent() 131, 145, 146
resizeGL() 208, 209
resizeImage() 420
ResizeTransaction 422
Ressources (intégration) 302
restore(), matrice de transformation 196
restoreState() (QMainWindow) 159
retranslateUi() 397, 398
retrieveData() 222, 223
right() 280, 283
rightSplitter 154
rmdir() 333
Rôle 231, 241
DisplayRole 97
EditRole 97
rollback() 313
rotate() 191, 196
row() (QModelIndex) 242
rowCount() 82, 244
rowSpan 148
rubberBandIsShown 132
rubberBandRect 132
run() 422
exécution multithread 408, 409, 412

S

Sauvegarde 85
save() 47, 52
Editor 162
fichier 60
matrice de transformation 196
module de rendu 188
opérations XML 370
saveAs() 47, 52, 60-61, 167
saveFile() 60, 165, 167
saveState() 463
QMainWindow 159
SAX (Simple API for XML) 359
SaxHandler 362
scale() 191
Script configure
-help 473
Qtopia 470
scroll() 139
QWidget 175
scrollTo() 239
SDI (Single Document Interface) 75
section 245
Sécuriser pour les scripts 456
seek() 294
SELECT 312
Select All (action) 52
selectColumn() 91
selectedId() 230
selectedRange() 89
Selection 224
selectionAsString() 220

selectRow() 91
Sémaphore
  emploi 414
  freeSpace 414
  usedSpace 414
sendDatagram() 354
sender() (QObject) 64
sendRequest() 346, 352
Séparateur 152
sessionFileName() 466
sessionId() 467
sessionKey() 467
setAcceptDrops() 221
setAllowedAreas() 158
setAutoDetectUnicode() 388
setAutoFillBackground() 126
setAutoRecalculate() 93
setBackgroundRole() 126, 136
setBit() 434
setBrush() 186
setByteOrder() 289
setChecked() 163
setClipRect() 138
setCodec() 295, 388
setCodecForCStrings() 389
setCodecForTr() 389
setColumnRange() 38, 68
  SortDialog 69
setCompositionMode() 198
setContext() (QShortcut) 172, 367
setCurrentCell() 80
  QTableWidget 66
setCurrentFile() 58, 60, 62, 165, 167
setCurrentIndex() 150-151
setCurrentRow() 151
setCursor() 132
setCurveData() 129
setData()
  champ de base de données 315
  index de modèle 316
  QMimeData 219
  réimplémentation 247
setDefault() 19
setDirty() 93, 99
setDuration() 193
setEditorData() 258, 260
setEditTriggers() 233
  QAbstractItemView 231
setEnabledOptions() (QPrintDialog) 205
setFeatures() (QDockWidget) 157
SetFixedSize (QLayout) 38
setFocus() 165
setFocusPolicy() 126
setFont() 186
setFormat() (QGLWidget) 209
setFormula() 84, 98, 298
setGeometry() (QWidget) 72
setHorizontalHeaderLabels() 232
setHost() 341
setIconImage() 110, 112
setImage() 421
  presse-papiers 224
setImagePixel() 116, 117
SetInterfaceSafetyOptions() 456
setItem() 97
  QTableWidget 84, 232
setItemPrototype() 81
  QTableWidget 98
setLayout() (QWidget) 9
setLayoutDirection() 395
setlocale() 395
setMargin() (QLayout) 147
setMessage() 408, 410
setMimeData() 224
setModal() 66
setModelData() 258
setMouseTracking() (QWidget) 117
setNum() 280
setOutputFormat() 199
setOverrideCursor() 132
setPen() 115, 186
setPenColor() 112
setPixel() (QImage) 117
setPixmap()
  presse-papiers 224
  QDrag 218
setPlotSettings() 127
setPrintProgram() (QPrinter) 199
setRadius() 455
setRelation() 323, 325
setRenderHint() 186
setRootNode() 252
setSelectionMode() 221
setShortcutContext() (QAction) 172
setShowGrid() 80
setSingleShot() 192
setSizePolicy() 110, 112, 126
setSizes() (QSplitter) 154
setSourceModel() 241
setSpacing() (QLayout) 147
setSpeed() 455
setStatusTip() 374
setStretchFactor() 154
setStyle() (QWidget) 130
setTabOrder() (QWidget) 22
setText()
  code XML 369
  presse-papiers 224
  QClipboard 88
  QLabel 424
  QPushButton 39
  widget Ticker 174
settings 235
setToolTip() 374
setupUi() 29, 31, 304
setValue() 9, 268, 276
setVisible() 36
setWidget() 155
setWidgetResizable() 156
setWindowIcon() (QWidget) 49
setWindowModified() 166
setWindowTitle() 62
  QWidget 8
setZoomFactor() 113
shear() 191

short 489
Show Grid 52, 71, 75
show() 65, 128, 165
ShowControls 451
showEvent() 173-174
showPage() 379
Signal 7
  clicked() 170, 237, 304-305, 348
  closeEditor() 260
  connected() 344-346
  connexion aux slots 23, 304
  contentsChanged() 166
  copyAvailable() 162
  currentCellChanged() 57
  currentRowChanged() 151
  description 22
  disconnected() 344, 349, 350
  done() 331, 340, 342
  editingFinished() 259
  error() 344, 349
  findNext() 65
  findPrevious() 65
  finished() 422
  itemChanged() 81
  listInfo() 335
  modified() 80, 85
  readyRead() 339, 342, 344, 347, 351
  requestFinished() 342
  requestStarted() 342
  stateChanged() 333
  timeout() 192
  toggled 36
  transactionStarted() 423
  triggered() 51, 400
  valueChanged() 9
  versus événement 170
  windowActivated() 161
signals 7, 17, 23
simplified() 282-283
size() (QFontMetrics) 174
sizeConstraint 38
sizeHint (propriété) 35, 124, 129, 150, 167, 465
  QWidget 20, 38, 57, 111
sizeof() 502
sizePolicy 112
sizes() (QSplitter) 269
skipRawData() 431
Slot 7
  accept() 36
  addRow() 233
  allTransactionsDone() 420
  close() 19
  closeAllWindows() 74
  commitAndCloseEditor() 259
  connectionClosedByServer() 349
  connectToHost 345
  connectToServer() 345
  connexion aux signaux 23, 304
  convertTo1Bit() 420
  convertTo3Bit() 420
  convertTo8Bit() 420
  copy() 88
  currentCdChanged() 324
  cut() 88, 163
  del() 90, 237
  description 22
  documentWasModified() 166
  edit() 306
  enableFindButton() 19, 21
  error() 349
  findClicked() 19, 21
  findNext() 91
  findPrevious() 92
  flipHorizontally() 420
  flipVertically() 420
  ftpDone() 331
  ftpListInfo() 335
  help() 379
  httpDone() 340
  insert() 237
  newFile() 51, 58, 73, 161
  on_browseButton_clicked() 304, 305
  on_convertButton_clicked() 304
  on_lineEdit_textChanged() 32
  on_objectName_signalName() 304
  open() 59
  openRecentFile() 64
  paste() 90
  processPendingDatagrams() 356
  recalculate() 92
  refreshTrackViewHeader() 327
  regExpChanged() 256
  reject() 36
  resizeImage() 420
  save() 60, 162
  saveAs() 61
  selectAll() 52
  sendRequest() 346
  setAutoRecalculate() 93
  setColumnRange() 38
  setFocus() 165
  setValue() 9
  setVisible() 36
  show() 165
  somethingChanged() 81, 85
  Spreadsheet 65
  spreadsheetModified() 57
  stopSearch() 348
  switchLanguage() 400
  updateMenus() 163
  updateOutputTextEdit() 306
  updateStatusBar() 57
  updateTableWidget() 347
  updateWindowTitle() 378
  zoomIn() 127
  zoomOut() 127
Slot() 7, 23
  mot clé 17

SmallGap 204
Socket
  QTcpSocket 341
  QtSslSocket 341
SolidPattern 187
somethingChanged() 81, 85, 93-94
sort() 68
  MainWindow 69
  Spreadsheet 81
SortDialog 68, 69
  setColumnRange() 69
Sorties 287
source() (QDragEnterEvent) 218
split() 282
  QString 90
Spreadsheet 49, 65
  clear() 58
  readFile() 60
  setFormula() 98
  sort() 68, 81
SpreadsheetCompare 68, 81, 94
  arbre d'héritage 79
spreadsheetModified() 57
sprintf() 279
Square 95
squeeze() 275
start()
  QDrag 218
  QProcess 305
  slot 455
startDrag() 217, 220
startElement() 360, 364
startOrStopThreadA() 410
startOrStopThreadB() 411
startPos 217
startsWith() 281
startTimer() 174
stateChanged() 333
static_cast 510
static_cast<T>() 64, 511
status() 290
statusBar() (QMainWindow) 56
StatusTipRole 242
std 125
STL (Standard Template Library) 63
  fichiers d'en-tête 525
Stockage
  de données en tant qu'éléments 82
  des caractères ASCII 490
  format 288
  local de thread 417
  XML 359
stop() 408
  exécution multithread 412
stopped 408, 411
stopSearch() 348
strippedName() 62, 165
strtod() 486
Style
  de widget 12, 130
  Plastique 12
  spécifique à la plate-forme 12
style() 130
submitAll() 315
sum() 104
supportsSelection() 225
Surcharge d'opérateur 512
switchLanguage() 400, 401
Symbole
  de préprocesseur 446
  de système de fenêtre 446
Synchronisation (des threads) 411
System() 394
Systèmes embarqués 469

T

Table de hachage 274
Tableau
  C++ 502
  d'octets 278
TableMimeData 224
tagName() 368
takeFirst() (QStringList) 297
Tas 498
tcpSocket 344
TeamLeadersDialog 238
terminate() 409
Tetrahedron 208
text() 27, 83, 137
  presse-papiers 224
  QLineEdit 66
  QTableWidgetItem 96
TextAlignmentRole 99, 242, 245, 256
TextArtDialog 437-438
TextArtInterface 437-438
TextColorRole 242, 256
Texte
  écriture 294
  lecture 294
textFromValue() 107
Thread
  communication 418
  consommateur 416
  création 408
  producteur 416
  setMessage() 408
  stop() 408
  synchronisation 411
  utilisation des classes Qt 423
ThreadDialog 410
Ticker 172
tidyFile() 298
tile() 164
timeout() 175, 192
timer 169, 175
timerEvent() 173, 175, 181, 452
title 34
toAscii() 283
toBack() 268
toCsv() 221
toDouble() 100, 280
toElement() 368
toggled() 36

toHtml() 221
toInt() 280
  QString 66, 107
toLatin1() 283, 387
toLongLong() 280
toLower() 281, 283
toolTip() 121
ToolTipRole 242
top() 266
toPage() (QPrinter) 205
toString() 99, 284, 395
toUpper() 281, 283
  QString 107
tr()
  déclaration avec Q_OBJECT 25
  internationalisation 392
  traduction de littéraux 18
Tracé de dessin 187
TrackDelegate 258, 323, 326
transaction() 313, 422
  ajout à la file d'attente 421
  ConvertDepthTransaction 422
  exécutions simultanées 314
  FlipTransaction 422
  ResizeTransaction 422
  SQL 313
transactionStarted() 422, 423
TransactionThread 420
  run() 423
translate() 191, 392-393
triggered() 51
trimmed() 282, 283
TripPlanner 344
TripServer 349
  incomingConnection() 349
truncate() 272
type() 284
  conversions 509
  MIME 215
  personnalisé de glisser 219
  primitifs 489
  QEvent 170
  représentation binaire 86
  valeur 514
TypeDef 509

U

UDP 329
  datagrammes 353
  écoute du port 355
  envoi et réception de datagrammes 353
uic 29, 32, 120
unget-Char() 294
unicode() 386-387
Unité de compilation 484
unlock() 411
unsigned 489
update()
  forcer un événement paint 113
  planifier un événement paint 136
  QRect 117
  QWidget 112
  viewport 93
updateGeometry() 113, 174
  QWidget 112
updateGL() 211
updateMenus() 163
updateOutputTextEdit() 306
updateRecentFileActions() 62, 74-75
updateRubberBand() 136
updateRubberBandRegion() 132
updateStatusBar() 57
  MainWindow 84
updateTableWidget() 348
updateWindowTitle() 378
UserRole 231
using namespace 486
uuidgen 457

V

Valeur 71
  bitsPerPixel 431
  compression 431
Valgrind 490
Validateur
  QDoubleValidator 31
  QIntValidator 31
  QRegExpValidator 31
value()
  champ de base de données 312, 314
  courbe 138
  itérateur STL 276
  map 274, 275
  valeur
    d'une cellule 100
    d'une map 275
valueChanged() 9
valueFromText() 107
values() 274
Variable
  globale 516
  mutable 97
  sur la pile 67
Variable d'environnement
  LD_LIBRARY_PATH 481
  PATH 5, 380
  QTDIR 122
  QWS_KEYBOARD 471
  QWS_MOUSE_PROTO 471
Variant 278
Vecteur
  d'éléments 264
  de coordonnées 122
  de QChar 279
  distances 248
  initialisation 277
  parcourir 265
Version de système d'exploitation 446

viewport() 155
  dessin 188
  OpenGL 209
  système de coordonnées 189
  transformation du painter 194
  update() 93
  widget 82
virtual 494
VNC (Virtual Network Computing) 472
void* 511
Vue
  QListView 229
  QTableView 229
  QTreeView 229

W

WA_DeleteOnClose 74, 162, 166
WA_GroupLeader 378
WA_StaticContents 117
wait() 411, 417
waitForDisconnected() 424
waitForFinished() 424
warning() (QMessageBox) 58, 70
WeatherBalloon 354
whatsThis() 121
WhatsThisRole 242
wheel 135
wheelEvent() 135
while 179
Widget
  ancrable 157
  bouton 40
    QCheckBox 40
    QPushButton 40
    QRadioButton 40
    QToolButton 40
  central 78
  classes 40
  compagnon 18
  conteneur 41
    multipage 41
    QFrame 41
    QTabWidget 41
    QToolBox 41
  d'affichage d'éléments 41
  d'entrée 43
  définition 4
  disposer 7, 28, 144
  HelpBrowser 377
  OvenTimer 191, 194
  palette de couleurs 115
  personnalisé 106
    créer 105
    et Qt Designer 108
    intégrer avec le Qt Designer 118
  Plotter 122
  style 12
  viewport 82
windowActivated() 161
windowModified 57, 62
Windows
  ActiveX 448
  identifier la version 446
  installer Qt 478
  Media Player 449
windowTitle 27
winEvent() 448
winEventFilter() 448
winId() 444
write() 429
  QIODevice 293
writeDatagram() 355
writeFile() 85
WriteOnly (QIODevice) 87
writeRawBytes() 290
writeSettings() 70, 71, 154

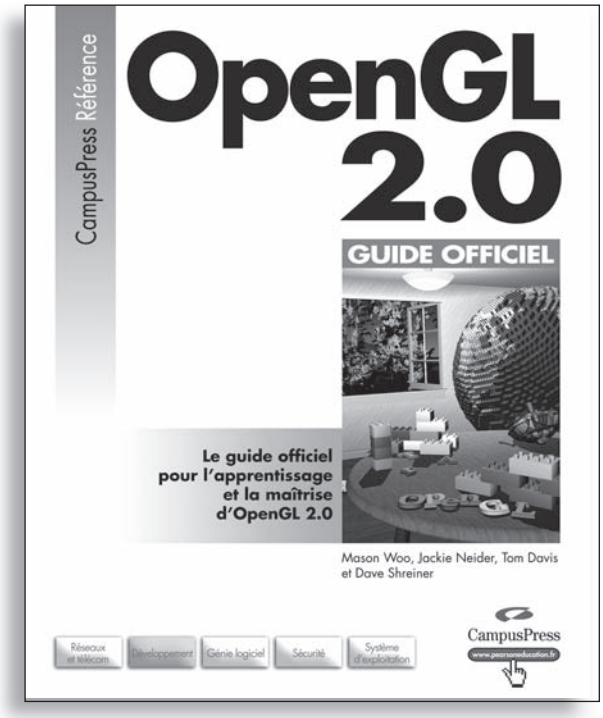
X

X Render 197
X11
  gestion de session 461
  installer Qt 479
x11Event() 448
x11EventFilter() 446
Xcode Tools 479
XML 359
  écriture 370
  lecture
    avec DOM 365
    avec SAX 360
xsm 467

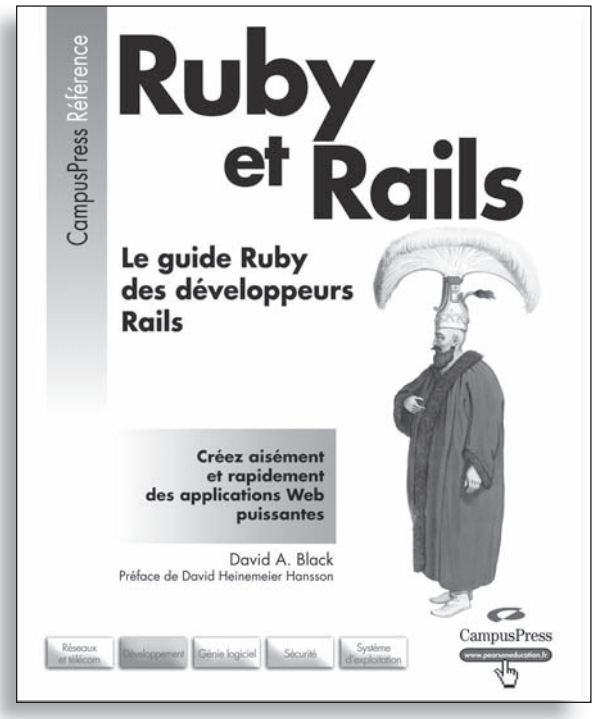
Z

Zone
  d'action 188
  déroulante 155
zoomFactor() 109, 113
zoomIn() 127, 134
zoomOut() 127
zoomStack 127

Chez le même éditeur



ISBN : 2-7440-2086-9 • Parution : 07/06 • Prix : 52 €



ISBN : 2-7440-2127-X • Parution : 12/06 • Prix : 49,90 €

CampusPress

Tous nos ouvrages sont disponibles sur www.pearsoneducation.fr

CampusPress Référence

Qt 4 et C++

Programmation d'interfaces GUI

Un ouvrage unique sur le développement d'interfaces graphiques avec la bibliohèque Qt, écrit par des spécialistes de Trolltech.

Grâce au framework Qt de Trolltech, vous pouvez créer des applications C++ de niveau professionnel qui s'exécutent en natif sous Windows, Linux/UNIX, Mac OS 10 et Linux intégré sans qu'aucune modification dans le code source soit nécessaire.

Ce guide complet vous permettra d'obtenir des résultats fantastiques avec la version la plus puissante de QT jamais créée : QT 4.1. En s'appuyant sur des exemples réalistes, il présente des techniques avancées sur divers sujets depuis le développement de l'interface graphique de base à l'intégration avancée de XML et des bases de données.

- Couvre l'ensemble des éléments fondamentaux de Qt, depuis les boîtes de dialogue et les fenêtres jusqu'à l'implémentation de la fonctionnalité d'une application
- Présente des techniques avancées que vous ne retrouverez dans aucun autre ouvrage, comme la création de plugins d'application et pour Qt, ou la création d'interfaces avec les API natives
- Contient des annexes détaillées sur la programmation C++/Qt destinée aux développeurs Java expérimentés

TABLE DES MATIÈRES

Partie I : Qt : notions de base
- Pour débuter
- Créer des boîtes de dialogue
- Créer des fenêtres principales
- Implémenter la fonctionnalité d'application
- Créer des widgets personnalisés

Partie II : Qt : niveau intermédiaire
- Gestion des dispositions
- Traitement des événements
- Graphiques 2D et 3D
- Glisser-déposer
- Classes d'affichage d'éléments
- Classes conteneur
- Entrées/sorties
- Les bases de données
- Gestion de réseau
- XML
- Aide en ligne

Partie III : Qt : niveau avancé
- Internationalisation
- Environnement multithread
- Créer des plug-in
- Fonctionnalités spécifiques à la plate-forme
- Programmation embarquée

Annexes
- Installer Qt
- Introduction au langage C++ pour les programmeurs Java et C#

A propos des auteurs

Jasmin Blanchette, responsable de la documentation chez Trolltech et développeur expérimenté travaille pour cette société depuis 2001. Il intervient comme éditeur de Qt Quaterly, le bulletin d'information technique de Trolltech.

Mark Summerfield travaille comme consultant et formateur spécialisé en C++, Qt et Python. Il a assumé la charge de responsable de la documentation chez Trolltech pendant presque trois ans.

Programmation

Niveau : Intermédiaire / Avancé
Configuration : Multiplate-forme

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

Officially Approved by Trolltech

ISBN : 978-2-7440-4092-4